Implementing a Reconfigurable Atomic Memory Service for Dynamic Networks*

Peter M. Musial[1]
(1) Department of Computer Science and Engineering
University of Connecticut, Storrs, CT 06269, USA

Alex A. Shvartsman[1,2]
(2) Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Abstract

Transforming abstract algorithm specifications into executable code is an error-prone process in the absence of sophisticated compilers that can automatically translate such specifications into the target distributed system. This paper presents a framework that was developed for translating algorithms specified as Input/Output Automata (IOA) to distributed programs. The framework consists of a methodology that guides the software development process and a core set of functions needed in target implementations that reduce unnecessary software development. As a proof of concept, this work also presents a distributed implementation of a reconfigurable atomic memory service for dynamic networks. The service emulates atomic read/write shared objects in the dynamic setting where processors can arbitrarily crash, or join and leave the computation. The algorithm implementing the service is given in terms of IOA. The system is implemented in Java and runs on a network of workstations. Empirical data illustrates the behavior of the system.

1. Introduction

Developing sophisticated distributed applications for extant distributed platforms presents a challenge, despite the availability of distributed middleware. Although middleware frameworks such as DCE, CORBA, and JINI support construction of systems from components, their specification capability is limited to the formal definition of interfaces and informal descriptions of behavior. These are not enough to support careful reasoning about the behavior of systems that are built using such services.

Specification of building blocks for distributed applications is an area of active research. However, even when specifications and algorithms are formally stated, deriving a distributed implementation from a specification is a laborious and error-prone process.

*This work is supported in part by the NSF Grants 9988304 and 0311368, and the NSF ITR Grant 0121277. The work of the second author is supported additionally by the NSF CAREER Award 9984778.

Atomic (linearizable) shared memory is a powerful abstraction that makes it easier to develop and reason about distributed applications that rely on shared data. To be useful as system building blocks, algorithms that implement the shared memory abstraction must be able to tolerate asynchrony and failures encountered in distributed settings.

In this paper we present a distributed implementation of a formally specified building block for a reconfigurable atomic data service. Our implementation is developed with the help of a framework that is designed to make the derivation process less error-prone. The system currently runs on a network-of-workstation. We include experimental results that explore the behavior of the service.

Background. The first scheme for emulating shared memory in message-passing systems by using replication and accessing majorities of time-stamped replicas was given in [14]. An algorithm for majority-based emulation of atomic read/write memory was presented in [2]. A generalization for multiple writers using quorums is given in [12].

A new approach to implementing atomic read/write objects for dynamic networks was developed by Lynch, Shvartsman, and Gilbert [10, 7]. Their service, called RAMBO, assumes a dynamic set of processors that can join and leave the computation, and fail. As processors come and go, reconfiguration of quorum systems is used to maintain atomicity. These algorithms are loosely coupled with the reconfiguration service that uses consensus, and they allow for read and write operation to complete even during reconfiguration, and even if reconfiguration fails to terminate. The algorithms implementing RAMBO are quite complex, however they rely on conventional point-to-point channels and tolerate arbitrary message latency and message loss.

The algorithms in [10, 7] are given using the Input/Output Automata (IOA) of Lynch and Tuttle [11]. There is a wealth of distributed algorithms that have been specified in the IOA notation [9]. In recent years several automated tools have been developed to reason about, and to simulate algorithms given as IOA [5, 1]. Current work is underway to implement automated code generator for IOA. However, deriving sophisticated distributed systems from formally-specified algorithms remains a manual process.

IEEE COMPUTER SOCIETY
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Contributions. An important consideration in formulating the RAMBO atomic shared memory service was that it could be employed as a building block in real systems. Our objective in this work is to develop a faithful implementation of RAMBO for a message-passing platform and using it as a basis for practical optimizations and for performance studies. Here we present our optimized implementation and a general methodology we developed for deriving such distributed systems from the formal Input/Output Automata (IOA) specifications. Our implementation was developed in Java and it runs on networks of Linux workstations. In more detail our contributions are as follows.

(1) Transforming abstract algorithm specifications into executable code is an error-prone process in the absence of sophisticated compilers that can automatically translate such specifications into the target distributed system. We present a framework for mapping algorithms specified as IOA to distributed programs. The framework includes a methodology that guides the software development process and a core set of functions needed in target implementations that reduce unnecessary software development. Each automaton in an IOA specification include *signature* that defines the interface of the automaton, *state* that defines the persistent private state of the automaton, and *transitions* that define, in precondition-effect style, the actions of the automaton that show how and when it interacts with its environment and alters its state. We define precise rules for deriving Java code for signature, state, and transitions. We also specify how the precondition evaluations are scheduled, how the effects of the actions are executed, and how to change the state of the automata atomically. *Compositionality* of automata is one of the strengths of the IOA formalism. We develop procedures for translating automata compositions that ensure that individual automata can fail individually without affecting the rest of the system. Finally, we provide components that streamline the development of systems where the individual automata communicate by means of asynchronous point-to-point channels.

(2) Using our methodology we implemented the distributed reconfigurable atomic memory algorithm [10, 7] and explored its behavior in the distributed setting. Our system, implemented in Java, currently runs a network of Linux workstations. The overall translation process is relatively straightforward due to our methodology.

We present selected optimizations of RAMBO, including optimized memory usage, reduced number of messages, and improved performance and fault-tolerance. Our implementation preserves the logical structure of the IOA specification of RAMBO, making it easier to develop and reason about the optimizations. All optimizations preserve the safety properties (atomicity) of the original system.

We present selected empirical results. This includes observations about the scalability of system throughput as we increase the number of nodes, and about the impact of reconfiguration on the system throughput.

Finally, our approach improves on the methodology of Cheiner and Shvartsman [3], who developed conversion rules for implementing IOA algorithms as C++ programs using MPI. Our approach substantially improves the traceability of the derived code to the source specifications and allows for the implementation of systems that tolerate node failures (this was not possible with [3]).

Structure of the extended abstract. In Section 2 we overview the IOA formalism. Section 3 describes the techniques used to translate the IOA code. In Section 4 we describe the RAMBO algorithm. In Section 5, we describe our implementation of RAMBO. In Section 6 we present empirical results. The conclusions is in Section 7.

2. Input/Output Automata Overview

The *Input/Output Automaton* [9, 6], or I/O Automaton, is a general model used for formal descriptions of asynchronous and distributed algorithms. The model provides a precise way of describing and reasoning about asynchronous interacting components in terms of labeled transition systems. We provide a concise description of the model and refer the reader to [9, 6] for more details.

An I/O automaton is a *state machine* in which the *transitions* are associated with named *actions*. The actions are classified as either *input*, *output* or *internal*. The input and output actions interact with the automaton's environment. The internal actions work on the automaton's local state. I/O automata code is given in a *precondition-effect* style. For each action, the code specifies the *preconditions* under which the action is permitted to occur, as a predicate on the automaton state, and the *effects* that occur as the result. The input actions are always enabled (i.e., the precondition clause of an input action is always *true*). The code in the effects clause is executed atomically to change the state.

Each automaton has a name and includes: (a) a *signature*, which defines the disjoint sets of the input, output, and internal actions, (b) a set of *states*, normally given as a collection of state variables, (c) a set of *start states* or *initial states* that is nonempty subset of the set of all states, (d) a *state-transition relation*, which contains *steps* or *transitions*, given as triples (state, action, state).

Consider some automaton A. A *transition* of automaton A is an element of *state-transition relation* that contains an element (s, π, s'), where s represents the *state* before *action* π occurred and s' represents the *state* after *action* π occurred. If A has transition (s, π, s') then we say that π is *enabled* in s. A *transition relation* π of automaton A is described in *precondition-effect* style, which groups together all transitions that involve a particular type of action into a single piece of code (see a simple example in Figure 1).

IEEE COMPUTER SOCIETY
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Domains: I, a set of processes M, a set of messages

States: $S \subseteq M$, the set of messages in the channel

Signature:

Input: $\mathsf{Snd}(m)_{i,j},\ m \in M$, const $i, j \in I$

Output: $\mathsf{Rcv}(m)_{j,i},\ m \in M$, const $i, j \in I$

Internal: $\mathsf{Lose}(m),\ m \in M$

Transitions:

Input $\mathsf{Snd}(m)_{i,j}$	Output $\mathsf{Rcv}(m)_{j,i}$	Internal $\mathsf{Lose}(m)$
Effect:	Precondition:	Precondition:
$S := S \cup \{m\}$	$m \in S$	$m \in S$
	Effect:	Effect:
	$S := S - \{m\}$	$S := S - \{m\}$

Figure 1. $Channel_{i,j}$ automaton specification.

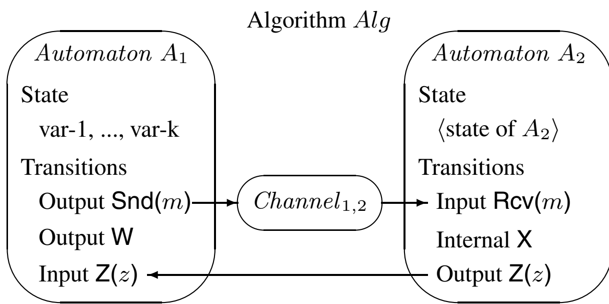

Figure 2. A skeleton of a specification is depicted. Algorithm Alg is the composition of three automata: A_1, A_2, and $Channel_{1,2}$. A_1 can send messages to A_2 via the channel, while A_2 communicates with A_1 using a rendezvous action Z. (Preconditions and effects are omitted.)

Automata Compositon. Complex systems can be constructed by *composing* automata representing individual system components. Composition identifies actions with the same name in different component automata. In a composition, when actions with same name from different automata are matched, we refer to such actions as *combination actions*; the rest of the actions are referred to as *regular actions*. 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 automata in a composition communicate by means of combination actions. When any component automaton performs a step involving a combination action π, so do all component automata that include the action π. (Please note that this presents an implementation challenge. The implementation has to make sure that the execution of π's transition relation over all participating automata is atomic. This is especially difficult when the component automata are mapped to distinct processors in a network.)

```
1:  class A1 {
2:   private static A1-state state;
3:   public static A1-state gs-state(A1-state S) {
4:    if(S == null) return state;
5:    state = S;
6:    return null;
7:   }
8:   public void A1( ) { }
9:   public static void init( ) {
10:   S = new A1-state( ); S.init();
11:   A1-output-Snd a1 = new A1-output-Snd( );
12:   (new Thread(a1)).start();
13:   A1-output-W a2 = new A1-output-W( );
14:   (new Thread(a2)).start();
15:   A-input-Z a3 = new A-input-Z( );
16:   (new Thread(a3)).start();
17:  }
18:  public static synchronized void scheduler(int ⟨thread-id⟩) {
19:   ⟨thread-id⟩.transition-code( );
20:  } }
```

Figure 3. The structure of a single automaton's translation.

The states and start states of the composition are vectors of states and start states of the component automata. (For examples see Figures 1 and 2.)

3. IOA Translation

The section presents a collection of rules developed to translate algorithms specified in IOA notation into programs in the Java language. These rules guide the programmer in faithfully encoding the structure of the algorithm and the interactions among the component automata in a way that help eliminate logic errors and to reduce redundant and repetitive work. Throughout this section we use the example given in Figure 2. The target language assumed here is Java 2 Platform Std. Ed. v1.4.0. The choice of Java was made because of the tools that this language provides.

IOA specification of an automaton is considered a *type* based on which a class is created and named $\langle$*automaton name*$\rangle$. Consider the automaton A_1 from Figure 2. The specification of A_1 is translated as a Java class named A_1 (see Figure 3). This corresponding Java class contains the following methods: (1) a state access method, which allows automaton's actions to read and update the state, (2) the *init* method, where the state variables are instantiated and the threads representing automaton's actions are started, and (3) the *scheduler* method controlling execution of local actions of this automaton. All public methods of this class constitute its interface and will be accessible to all inheriting classes.

Translating Automaton's Signature. The *signature* of an automaton defines its interface (see Figure 6 for a detailed example). For the automaton A_1 in Figure 2, the signature (that is not shown in the figure) consists of the three action definitions: output $\mathsf{Snd}(m)$, output W, and in-

IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

```
1: class A1-input-Z extends A1 implements Runnable {
2:   public void A1-input-Z( ){ }
3:   public static void init( ) {
4:     // If needed initiate any local variables.
5:   }
6:   public void run( ) {
7:     while(true) scheduler(A1-input-Z);
8:   }
9:   private void transition-definition( ) {
10:    A-state copy = gs-state(null); // get state
11:    if (⟨Z-preconditions⟩) {
12:      ⟨Z-effect⟩ // The effect code goes here.
13:      gs-state(copy) // update state
14:    } } }
```

Figure 4. Action signature and transition translation.

put $\mathsf{Z}(z)$. In the translation, each action in the signature is treated as a type, based on which a Java class is created and named ⟨*automaton*⟩-⟨*kind*⟩-⟨*action name*⟩, e.g., A_1-input-Z, see Figure 4. These classes are children of ⟨*automaton name*⟩ class, hence they inherit the parent's interface, including methods to access the state, and the schedule method.

Translating Automaton's State Variables. The state variables of an automaton are encapsulated in a class named ⟨*automaton*⟩*-state*, e.g., for the automaton A_1, it is called A_1*-state*. This class is used to create a private instance of the automaton's state.

Each variable of A_1*-state* is declared as private and is coupled with a dedicated public get/set method. Lastly, the state class contains an initialization method for the state variables. This method is called by the automaton after its state is instantiated. (Of course the translation must be careful to using concrete data types that have finite representation, where the abstract data types may have infinite representation.)

Translating Automaton's Transitions Transitions define the semantics of the actions in the signature. Each transition is encoded as a private method of ⟨*automaton*⟩-⟨*kind*⟩-⟨*action*⟩, Figure 4, lines 10 to 14. An action is a state transition that occurs as a result of executing the transition definition code for the specific values of parameters (if any). This task is performed by a Java thread created from ⟨*automaton*⟩-⟨*kind*⟩-⟨*action*⟩ class. We refer to it as the *action thread*. All action threads are initiated and started in ⟨*automaton*⟩ class.

IOA transitions are defined in the precondition-effect style. The predicate of the precondition is converted to the corresponding Java code. The effect is given as sequential code, and the translation is straightforward. Following a call to the assigned transition method, the action thread reads and checks the state of the automaton, Figure 4, line 10. If the preconditions are satisfied in the current state (line 11), then the action is enabled, and the effects code of the transition definition is executed (line 12). If the preconditions are not satisfied then the action thread returns without changing the state. This process is repeated until stopped.

Scheduling. We implement a control mechanism, called *schedule*, that enables the action threads to run. To ensure that the state is changed atomically, the mechanism allows only a single action thread to get/set the automaton's state and to execute its transition definition. The schedule is implemented by placing Java's *synchronize* keyword in front of the schedule method of ⟨*automaton*⟩ class. After a thread enters the schedule method it makes a call to the corresponding transition definition method.

The *synchronize* mechanism provides a semaphore that allows locking and unlocking of the method by a single thread. A FIFO queue is associated with the lock. When multiple threads contest for the semaphore, only one succeeds and the rest is placed on the queue.

Note that whenever a thread is ready to execute, it may have to wait at most once for any other thread that previously became ready to execute. Since the number of threads is fixed at compile-time, our mechanism implements a fair scheduling policy. The implementation can be easily refined to introduce different scheduling policies.

The IOA formalism makes a nondeterministic choice of a single action among all enabled actions. In our implementation, the schedule chooses the first action thread corresponding to an enabled action from the *synchronized* method's queue. (An execution of a thread corresponding to a disabled action does not change state, and only incurs a modest computation cost.) Thus our implementation restricts the set of possible executions—of course this preserves the safety properties of the source algorithm.

Translating Local Combination Actions. We now discuss the translation of *local* action combination pairs, that is the combinations in the composition of automata that are mapped onto the same network node. This is an important part of the overall translation. The challenges here are to: (a) maintain the atomicity of the update of the combined state by the effects of the actions of the composed automata, and (b) ensure that a failure or a block of a single automaton does not cascade through the system by damaging or blocking other automata.

The implementation uses a "delicatessen" with "ticket numbers" paradigm, where the output action thread plays the role of the "customer" and the input action thread plays the role of the "server." Here the "transaction" ensures that the respective automata states are changed atomically. Timeouts are used to ensure that the implementation does not block when individual automata crash (or block for whatever reason). (The details are given in [13].)

If neither the server, nor the customer fail, they will synchronize in the delicatessen and correctly update their respective states. If either the customer or the server fail to synchronize, neither state will be updated. If either of them

IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

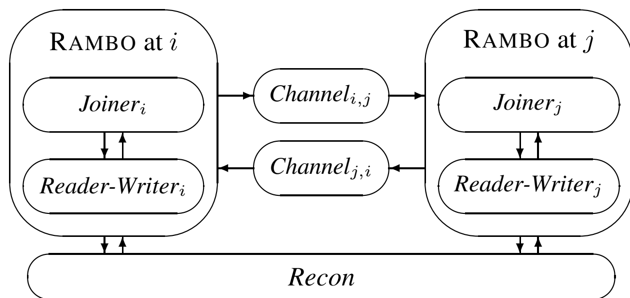

Figure 5. RAMBO component architecture depicting the $Joiner$ and $Reader\text{-}Writer$ automata at nodes i and j, the $Channel$ automata, and the $Recon$ service.

crash, the state of the crashed automaton is lost (this is not a problem). What is important here is that neither the crash of an automaton, nor the failure to "transact," cause blocking of any other enabled action.

Note, this method can be extended to the composition of more then two automata (when the participating automata are mapped onto the same node).

Remote Combination Action Pairs. Translation of a *remote* combination action pair, that is the combination in automata that reside on different network nodes, uses the same logic as its *local* equivalent. All communication is implemented using *timeout-send*, *timeout-receive* and *nonblocking* network calls. The *nonblocking* receive call is implemented in Java using a minimum timeout value on a socket of one millisecond. The "delicatessen" handshake mechanism is replaced by the TCP-handshake mechanism.

Communication via Channel Automata. Point-to-point communication in IOA is normally modelled as *channel automata*. Figure 1 and 2 illustrates a complete IOA specification of a channel from some fixed process i to some fixed process j ($Channel_{i,j}$). The channel delivers only the messages that were sent. It does not order messages, and it can lose messages. The channel has three actions. The input Snd action accepts a message at processor i, the output Rcv action delivers a message to processor j, and the internal lose action forgets a message. Note that if one wants to model a lossless channel, we simply remove the Lose action. If we want a FIFO channel, we use a queue instead of a set to model the state of the channel.

There are various types of channels, for example, a channel can be specified to be FIFO (or unordered), to be lossless (or lossy), etc. When such channels are implemented, the choice of the most appropriate underlying communication medium must be carefully assessed.

Our derivation of channel implementations from IOA specifications follows the methodology proposed in [5].

4. Description of the RAMBO Algorithms

We now present the algorithms implementing the Reconfigurable Atomic Memory Service [10, 7], called RAMBO and specified in terms of I/O automata. RAMBO implements atomic read/write shared memory in a dynamic network setting, in which participants may join or fail during the course of computation. In order to achieve fault tolerance and availability, RAMBO replicates objects at several network locations. In order to maintain memory consistency in the presence of small and transient changes, the algorithm uses *configurations*, each of which consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. The quorum intersection property requires that every read-quorum intersect every write-quorum. To accommodate larger and more permanent changes, the algorithm supports *reconfiguration*, by which the set of members and the sets of quorums are modified. Such changes do not cause violations of atomicity. Any quorum configuration may be installed at any time – no intersection requirement is imposed on the sets of members or on the quorums of distinct configurations.

The RAMBO algorithm for a processor in a network consists of two kinds of automata: (i) *Joiner* automata, which handle join requests, (ii) *Reader-Writer* automata, which handle read and write requests, and manage configuration upgrades. These component communicate by means of of point-to-point *Channels*. (Observe that the implementation of the *Channel* automata satisfies the RAMBO assumptions about the channel, specifically that messaging is asynchronous and that messages are not corrupted.) The algorithm also interacts with an underlying *Recon* service that emits a sequence of configurations based on the reconfiguration requests from the environment. This high-level architecture is given in Figure 5.

To avoid a complete restatement in this extended abstract, we refer the reader to [10, 7] for the description of algorithms and their specifications.

The external signature for the RAMBO algorithm appears in Figure 6. The algorithm is specified for a single memory location, and extended to implement a complete shared memory. A client at node i uses the join_i action to join the system. After receiving a $\mathsf{join\text{-}ack}_i$, the client can issue read_i and write_i requests, which results in $\mathsf{read\text{-}ack}_i$ and $\mathsf{write\text{-}ack}_i$ responses. The client can issue a recon_i request to install a new configuration. Finally, the fail_i action is used to model node i failing.

The read and write operation consist of two phases. During each phase, quorums of processors are accessed, requiring a round-trip message delay. Assuming that there is some upper bound d on message delays, this means that each phase can take as little as $2d$ time. In fact the upper bound latency of $4d$ was shown for read and write operations in the absence of reconfiguration [10]. The analysis

IEEE COMPUTER SOCIETY
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Domains:
I, a set of processes; V, a set of legal values
C, a set of configurations, each consisting of members, read-quorums, and write-quorums

Input:
$\mathsf{join}(\mathsf{rambo}, J)_i,\ J \subseteq I - \{i\}, i \in I,$ such that if $i = i_0$ then $J = \emptyset$
$\mathsf{read}_i,\ i \in I$
$\mathsf{write}(v)_i,\ v \in V, i \in I$
$\mathsf{recon}(c, c')_i,\ c, c' \in C, i \in members(c), i \in I$
$\mathsf{fail}_i,\ i \in I$

Output:
$\mathsf{join\text{-}ack}(\mathsf{rambo})_i, i \in I$
$\mathsf{read\text{-}ack}(v)_i, v \in V, i \in I$
$\mathsf{write\text{-}ack}_i, i \in I$
$\mathsf{recon\text{-}ack}(b)_i, b \in \{ok, nok\}, i \in I$
$\mathsf{report}(c)_i, c \in C, i \in I$

Figure 6. RAMBO: External signature

of operation latency with ongoing reconfiguration is much more complicated. However, it can be shown that for certain benign steady-state settings where processors do not fail too quickly after becoming members of new configurations, and where the configuration upgrades keep up with reconfigurations, the operation latency is at most $8d$ [10, 7].

5. RAMBO Implementation and Optimization

Starting with the IOA code for RAMBO, the implementation skeleton is readily produced using the rules outlined in the previous section. This activity is straightforward and is amenable to future automatic translation. The main remaining activity is to faithfully translate the transition definition code that consists of the sequential code for the preconditions and effects.

The RAMBO algorithm is loosely coupled with the consensus algorithm that is used in reconfiguration (in fact, read and write operations terminate even if consensus fails to terminate [10, 7]). Given that any consensus algorithm can be used with RAMBO, we do not discuss this further. In the remainder we discuss several optimizations of RAMBO. (For all algorithmic changes, we formally modified the IOA specifications from [10, 7] and reasoned about the correctness of the refined algorithms; for lack of space the IOA code is not presented here.)

Memory Use Optimizations. IOA algorithms often use abstract objects, e.g. sets, that are infinite, that may grow without bound, or that may be needlessly replicated for convenience. In many cases these objects can be redefined so that only a modest space is needed to represent them. We have done this in RAMBO, of course making sure that the semantics is preserved.

Another type of optimization (currently in progress) addresses the situation where state components are replicated in multiple automata that are mapped onto the same network node. This is the case with the local knowledge about the configurations. For example, when the *Recon* automaton at node i learns about a new configuration, the algorithm allows it to immediately inform the *Reader-Writer*$_i$ automaton. This requires replicating memory and a somewhat costly local combination. Using a shared data structure for both automata, the whole process is streamlined. The correctness of this optimization is immediate.

Managing the Number of Messages. To propagate information through the system, the participating nodes send gossip messages, involving all-to-all exchange of information. When gossip is not throttled, the unnecessary network congestion may adversely affect performance. RAMBO allows for gossip messages to be sent non-deterministically at arbitrary times. The correctness of the algorithm does not depend on the timing of these messages.

Our implementation removes the gossip non-determinism: each node sends such messages as soon as it is locally known that these message need to be sent to process a new operation (e.g., read or write), or in response to a new operation in progress. Otherwise the node sends these messages at some predetermined intervals (as in the performance evaluation in [10, 7]).

Our next optimization is aimed at reducing network communication when the number of nodes is large, and read and write operations by the nodes that do not belong to active configurations are infrequent.

We divide the set of nodes participating in RAMBO into two groups: those that belong to some active configurations, referred to as *owners*, and the rest, called the *clients*. We allow only the owners to send gossip messages (of course both the owners and the clients are allowed to perform read and write operations).

Claim 5.1 *Restricting gossip preserves correctness (safety) of* RAMBO.

The quantitative advantage of this optimization is as follows. Suppose n processors join the system, and r of them are owners. Without the optimization, for each complete gossip round (assuming for the moment some global clock), the total of n^2 gossip messages are sent. With the optimization, only r^2 gossip messages are sent per gossip round. Since $r \leq n$, substantial savings can be realized given the quadratic nature of gossip.

(We also developed an optimization to reduce the size and number of gossip messages [8]. Gossip messages include the set of processor identifiers that joined the system. This information need not be gossiped to the nodes that have previously received it. The results in Section 6 do not take advantage of this optimization.)

IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

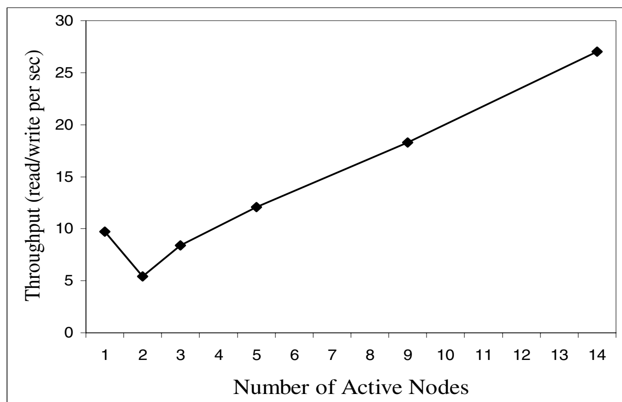

Figure 7. System throughput of Read/Write operations

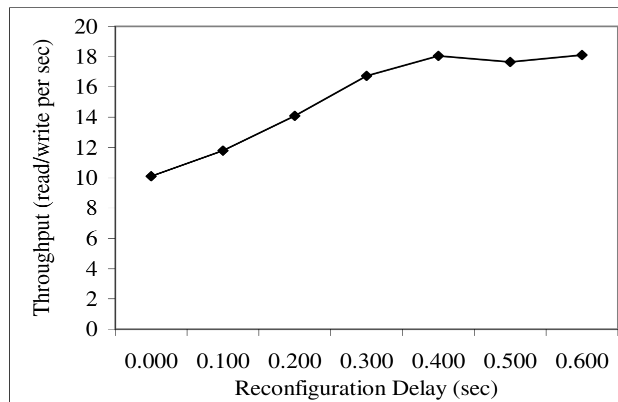

Figure 8. System throughput of Read/Write operations

Improving Performance and Fault-Tolerance. When a node is executing a read or a write operation in RAMBO algorithms, it is required that at least one quorum from each locally known active configuration is successfully contacted. The active configurations are fixed at the start of the operation. However, during the operation, it may be the case that some older configurations are upgraded and that the members of the old configurations become slow, leave the system, or fail. It may also be the case that some response messages are lost. If any of this occurs, then the operation may be delayed or even blocked.

Our optimization allows for slow or blocked operations (the distinction is not knowable locally) to restart in the state where the locally known collection of active configurations is potentially more up to date. This is done by imposing an adaptive time-out on read and write operations based on the prior performance of operations. Operations that are interrupted after the time-out are restarted using the latest known active configurations. This occurs transparently to the clients of the service.

Claim 5.2 *Restarting a read or a write operation preserves correctness (safety) of* RAMBO.

We believe that this type of optimization deserves future attention and we plan to identify other interesting scenarios where blocking is not necessary to preserve atomicity.

6. Empirical Results

We implemented and evaluated RAMBO on the dedicated cluster of fourteen Linux workstations connected via a 100 Mbps Ethernet network switch. The results in this section represent averages of the observed behavior over a series of instrumented runs. The significance of the observations is in the relative performance characteristics and behavior patterns of the system—several machines used in the experiments are quite old, some running at 100 MHz.

Throughput and Scalability. Figure 7 shows how the overall throughput of read/write operations of our implementation as the function of the number of participating nodes. In this series of runs, fixed and infrequent reconfigurations are performed by two dedicated reconfigurers, with exception of the first run since only one processor is used and it does not perform any reconfigurations. In all other runs the quorums in the configurations consist of processor majorities, thus the replication is increased linearly in the number of participating processors. This means that the reconfiguration "burden" is increased, as is the need to contact a linear number of replicas for each read or write, however the distribution and availability also increases substantially. All processors used in the runs are performing read and write operations (including processors designated as reconfigurers). Please note that since reconfiguration is present the configuration upgrade is enabled and runs concurrently with read/write and reconfiguration operations.

The throughput of read/write operations is relatively high when using a single node—here no network delays are incurred. The throughput drops when the second node is introduced. This is attributed to messaging latency. As further nodes are introduced, throughput is increased approximately linearly with the number of nodes. Thus the system throughput scales well, at least up to the fourteen participating processors.

Throughput and Reconfiguration. We now describe the series of experiments that explored the impact of the reconfiguration frequency on the throughput of read and write operations. Here we vary the rate at which new configurations are submitted to the system.

The result is shown in Figure 8. The vertical axes shows the throughput of read/write operations and the horizontal axes shows the delay between the completion of one reconfiguration to the start of another. The experiment involved nine machines performing read/write operations (without delay), and one machine acting as the submitter of reconfiguration requests (it is of course possible to have more then one machine submitting reconfiguration requests, but we use one to better control the rate of the reconfiguration in

IEEE COMPUTER SOCIETY
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

the experiments). Again, since new configurations are continuously proposed and installed the configuration upgrade operation is enabled and runs concurrently with read/write and reconfiguration operations.

The experiment was conducted for seven different reconfiguration rates, i.e., we changed the amount of wait time between the completion of one reconfiguration and start of another. An important observation is that at zero delay the consensus service itself limits the rate of reconfiguration.

The results predictably indicate that frequent reconfiguration negatively impact system throughput. Of course network congestion is a possible reason, however, the analysis of RAMBO [10, 7] predicts that in a steady state with ongoing reconfigurations, read and write operations take up to twice as long. This is because during each phase a new configuration may be discovered, and this requires that a quorum in the new configuration is accessed. It is also the case that if multiple new configurations are submitted at about the same time, only one "winning" configuration becomes the new next configuration. Thus in most settings, new configurations will be installed only once or twice concurrently with the ongoing read or write operations.

Results collected thus far illustrate the behavior of the RAMBO implementation on up to fourteen nodes in the LAN setting. In the future we plan to test the system in a WAN setting using a much larger number of nodes. This will give us a better idea of the behavior and scalability of the implementation.

On the Expense of Reconfiguration. In the experiments described thus far, we measured the throughput of read and write operations in the presence of reconfigurations. It is also interesting to compare this to the behavior of the system where no reconfigurations occur. We have conducted a series of experiments similar to those described in Section 6, but using a single configuration (the quorums are again majorities). Interestingly, the results are very close to those reported in Figure 7. This suggests that the cost of reconfiguration is not particularly significant in our setting. This also helps explain why the graph in Figure 8 is nearly "flat." It will be interesting to re-evaluate the expense of reconfiguration in the settings with faster machines.

7. Discussion

Developing dependable fault-tolerant distributed systems continues to be a challenge. Many systems are lacking formal specifications and precisely stated guarantees regarding fault-tolerance and performance. Even when such specifications are available, developing reliable systems that are faithful to the specifications is a laborious and error-prone process. We presented a collection of rules for developing fault-tolerant distributed systems that are formally specified using the Input/Output Automata (IOA) formalism. These rules guide developers in producing dependable code from formal specifications, and reduce the possibility of logic errors that are often made during translation. We used our methodology in developing a fault-tolerant implementation of a distributed atomic memory system, RAMBO [10, 7]. The resulting implementation maintains the modularity of the original formal specification. This substantially improves the traceability of the algorithmic properties.

Acknowledgements The authors thank Nancy Lynch, Steve Garland, and Seth Gilbert for many discussions of the implementation issues.

References

[1] IOA toolkit documentation and code: http://theory.lcs.mit.edu/tds/ioa/

[2] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message Passing Systems. *J. of the ACM* 42(1):124–142, 1996.

[3] O.M. Cheiner and A.A. Shvartsman. Implementing an eventually-serializable data service as a distributed system building block. *Networks in Distributed Computing, DIMACS Series*, 45:43–71, 1999.

[4] B. Englert and A.A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proc. Int. Conference on Distributed Computer Systems*, pp. 454–463, 2000.

[5] S.J. Garland and N.A. Lynch. Using I/O automata for developing distributed systems. *Foundations of Component-Based Systems*, pp. 285–312, Cambridge Univ. Press, 2000.

[6] S.J. Garland, N.A. Lynch, and M. Vaziri. *IOA: A language for specifying, programming, and validating distributed systems.* User and Reference Manual. LCS, MIT, 2001

[7] S. Gilbert, N. Lynch, and A.A. Shvartsman. RAMBO II: rapidly reconfigurable atomic memory for dynamic networks. In *Proc. of the International Conference on Dependable Systems and Networks*, pp. 259–268, 2003.

[8] C. Georgiou, P.M. Musial, and A.A. Shvartsman. Reconfigurable Atomic Memory for Dynamic Networks with Graceful Departures and Incremental Gossip. http://www.cse.uconn.edu/~piotr/publications/GMS03.ps

[9] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[10] N. Lynch and A.A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of 16th Int. Symposium on Distributed Computing, pp. 173–190, 2002.*

[11] N.A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report LCS/TR387, MIT, 1987.

[12] N. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proc. of 27th Int-l Symp. on Fault-Tolerant Comp.*, pp. 272–281, 1997.

[13] P.M. Musial and A.A. Shvartsman. Synchronization issues in implementing a distributed memory service. Manuscript, 2004.

[14] E. Upfal and A. Wigderson, How to share memory in a distributed system. *J. of the ACM* 34(1):116–127, 1987.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE