Highly Concurrent Logically Synchronous Multicast

Kenneth J. Goldman

July 3, 1989

Abstract

We define the *logically synchronous multicast* problem, which imposes a natural and useful structure on message delivery order in an asynchronous system. In this problem, a computation proceeds by a sequence of *multicasts*, in which a process sends a message to some arbitrary subset of the processes, including itself. A logically synchronous multicast protocol must make it appear to every process as if each multicast occurs simultaneously at all participants of that multicast (sender plus receivers). Furthermore, if a process continually wishes to send a message, it must eventually be permitted to do so.

We present a highly concurrent solution in which each multicast requires at most $4|S|$ messages, where S is the set of participants in that multicast. The protocol's correctness is shown using a remarkably simple problem specification stated in the I/O automaton model. We also show that implementing a wait-free solution to the logically synchronous multicast problem is impossible.

The author is currently developing a simulation system for algorithms expressed as I/O automata. We conclude the paper by describing how the logically synchronous multicast protocol can be used to distribute this simulation system.

Keywords: distributed computing, distributed algorithms, concurrency, synchronization, logical time, discrete event simulation, input/output automata

©1989 Massachusetts Institute of Technology, Cambridge, MA 02139

This research was supported in part by the National Science Foundation under Greant CCR-86-11442, by the Office of Naval Research under Contract N00014-85-K-0168, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, and by an Office of Naval Research graduate fellowship.

1 Introduction

We consider a set of n processes in an asynchronous system whose computation proceeds by a sequence of *multicasts* (or *partial broadcasts*). In each multicast, a process u sends a message m to an arbitrary subset S of the processes (including u). We say that a protocol solves the *logically synchronous multicast* problem if it guarantees the following conditions:

(1) Processes receive all messages in the same relative order. (Suppose messages m and m' are both sent to processes u_1 and u_2. If u_1 receives m before m', then u_2 does also, even if m and m' were sent by different processes.)

(2) If process u sends message m, it receives no messages between sending and receiving m.

(3) If process u continually wishes to send a message, then eventually u will send a message.

We may informally summarize the first two conditions by saying that it appears to all processes as if each multicast occurs simultaneously at all of its participants (sender plus receivers). Hence, the name *logically synchronous multicast.* Note that the hypothesis of the third condition does not require that u continually wish to send the *same* message, but only *some* message. This is a technical point that will be of importance later.

The problem lends itself to a highly concurrent solution, since any number of multicasts with disjoint S sets should be able to proceed independently. Likewise, one would expect that the communication costs of an algorithm to solve this problem would be independent of n. We present a solution to this problem that takes advantage of the concurrency inherent in the problem and requires at most $4|S|$ messages per multicast.

The strong properties of message delivery order imposed by the problem would make a fault-tolerant solution highly attractive for many applications. However, the properties of the message delivery order are strong enough to make a fault-tolerant solution impossible! By a reduction to distributed consensus, we show that there exists no wait-free solution to the logically synchronous multicast problem.

Various other approaches to ordering messages in asynchronous systems have been studied. Lamport [La] uses logical clocks to produce a total ordering on messages. Birman and Joseph [BJ] present several types of fault tolerant protocols. Their ABCAST (atomic broadcast) protocol guarantees that broadcast messages are delivered at all destinations in the same relative order, or not at all. Their CBCAST (causal broadcast) protocol provides a similar, but slightly weaker, ordering guarantee to achieve better performance. The CBCAST guarantees that if a process broadcast sends a message m based on some other message m' it had received earlier, then m will be delivered after m' at all destinations they share.

Like ours, the protocols of both [La] and [BJ] deliver messages to the destination processes according to some global ordering. However, these protocols do not solve the logically synchronous multicast problem because they allow messages to "cross" each other. That is, in their protocols a process u may send a message m and some time later receive a message ordered before m. Our problem requires that when a process u sends a message m, it must have "up to date" information, meaning that it has already received all messages destined for u that are ordered before m. (See Condition (2) above.)

Multiway handshaking protocols have been studied extensively for implementations of CSP [Ho] and ADA [DoD] (for example, see [Ba1] and [Ba2]). These protocols enforce a very strict ordering on system events, and therefore achieve less concurrency (than ours and the others mentioned above). This is necessary because the models of CSP and ADA permit processes to block inputs. Since a decision about whether or not to accept an input may depend (in general) on earlier events, each

process can only schedule one event (input or output) at a time. Our problem permits processes to schedule multiple input events at a time.

One interesting feature of our problem is that it lies in between the two general approaches described above. It permits more concurrency than the multiway handshaking protocols, yet imposes a strong, useful structure on the message delivery order.

Other related work includes papers by Awerbuch [Aw] and Misra [Mi], which study different problems in the area of simulating synchronous systems on asynchronous ones. In both cases, the computational models being simulated are very different from ours, but it is interesting to note that some of Misra's techniques, particularly those for breaking deadlock, can be applied to our problem.

The remainder of the paper is organized as follows. Section 2 provides a brief introduction to the I/O automaton model. In Section 3, we present the architecture of the problem and a statement of correctness in terms of the model. In Section 4, we formally present the algorithm using the I/O automaton model. In Sections 5 and 6, we sketch a formal correctness proof and present the message and time complexities. We prove in Section 7 that there exists no wait-free solution to the logically synchronous multicast problem.

The author is currently developing a simulation system for algorithms expressed as systems of I/O automata. The logically synchronous multicast problem was motivated by a desire to distribute the simulation system on multiple processors using asynchronous communication. We conclude the paper by describing how the logically synchronous multicast protocol can be used to achieve such a distributed simulation.

2 The Model

The logically synchronous multicast problem statement, protocol, and correctness proof are all formally stated using the I/O Automaton model [LT1, LT2]. We have chosen this model because it encourages precise statements of the problems to be solved by modules in concurrent systems, allows very careful algorithm descriptions, and can be used to construct rigorous correctness proofs. In addition, the model can be used for carrying out complexity analysis and for proving impossibility results. The following introduction to the model is adapted from [LT3], which explains the model in more detail, presents examples, and includes comparisons to other models.

2.1 I/O Automata

I/O automata are best suited for modeling systems in which the components operate asynchronously. Each system component is modeled as an I/O automaton, which is essentially a nondeterministic (possibly infinite state) automaton with an action labeling each transition. An automaton's actions are classified as either 'input', 'output', or 'internal'. An automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action. An automaton is said to be *closed* if it has no input actions; it models a closed system that does not interact with its environment.

Formally, an *action signature* S is a partition of a set $acts(S)$ of *actions* into three disjoint sets $in(S)$, $out(S)$, and $int(S)$ of *input actions*, *output actions*, and *internal actions*, respectively. We denote by $ext(S) = in(S) \cup out(S)$ the set of *external actions*. We denote by $local(S) = out(S) \cup int(S)$ the set of *locally-controlled actions*. An I/O automaton consists of five components:

- an action signature $sig(A)$,
- a set $states(A)$ of *states*,

- a nonempty set $start(A) \subseteq states(A)$ of *start states*,
- a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for every state s' and input action π there is a transition (s', π, s) in $steps(A)$, and
- an equivalence relation $part(A)$ partitioning the set $local(A)$ into at most a countable number of equivalence classes.

The equivalence relation $part(A)$ will be used in the definition of fair computation. We refer to an element (s', π, s) of $steps(A)$ as a *step* of A. If (s', π, s) is a step of A, then π is said to be *enabled* in s'. Since every input action is enabled in every state, automata are said to be *input-enabled.* This means that the automaton is unable to block its input.

An *execution* of A is a finite sequence $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of A for every i and $s_0 \in start(A)$. The *schedule* of an execution α is the subsequence of α consisting of the actions appearing in α. The *behavior* of an execution or schedule α of A is the subsequence of α consisting of *external* actions. The same action may occur several times in an execution or a schedule; we refer to a particular occurrence of an action as an *event.*

2.2 Composition

We can construct an automaton modeling a complex system by composing automata modeling the simpler system components. When we compose a collection of automata, we identify an output action π of one automaton with the input action π of each automaton having π as an input action. Consequently, when one automaton having π as an output action performs π , all automata having π as an action perform π simultaneously (automata not having π as an action do nothing).

Since at most one system component controls the performance of any given action, we cannot allow A and B to be composed unless the locally controlled actions of A and B form disjoint sets. Also, we require that each action of a composition must be an action of only finitely many of the composition's components. If A is the composition of a set Q of automata, then $int(\mathcal{A}) = \bigcup_{\mathcal{A}' \in Q} int(\mathcal{A}')$, $out(\mathcal{A}) = \bigcup_{\mathcal{A}' \in Q} out(\mathcal{A}')$, and $in(\mathcal{A}) = \bigcup_{\mathcal{A}' \in Q} in(\mathcal{A}') - \cup_{\mathcal{A}' \in Q} out(\mathcal{A}')$. Given an execution $\alpha = \vec{s_0}\pi_1\vec{s_1}\ldots$ of A, let $\alpha|A_i$ be the sequence obtained by deleting $\pi_j\vec{s_j}$ when π_j is not an action of A_i and replacing the remaining $\vec{s_j}$ by $\vec{s_j}[i]$.

2.3 Fairness

Of all the executions of an I/O automaton, we are primarily interested in the 'fair' executions — those that permit each of the automaton's primitive components (i.e., its classes) to have infinitely many chances to perform output or internal actions. The definition of automaton composition says that an equivalence class of a component automaton becomes an equivalence class of a composition, and hence that composition retains the *essential* structure of the system's primitive components. In the model, therefore, being fair to each component means being fair to each equivalence class of locally-controlled actions. A *fair execution* of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of $part$(A):

1. If α is finite, then no action of C is enabled in the final state of α.
2. If α is infinite, then either α contains infinitely many events from C, or α contains infinitely many occurrences of states in which no action of C is enabled.

We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A, and we denote the set of fair behaviors of A by $fairbehs(A)$.

2.4 Problem Specification

A 'problem' to be solved by an I/O automaton is formalized essentially as an arbitrary set of (finite and infinite) sequences of external actions. An automaton is said to *solve* a problem P provided that its set of fair behaviors is a subset of P. Although the model does not allow an automaton to block its environment or eliminate undesirable inputs, we can formulate our problems (i.e., correctness conditions) to require that an automaton exhibits some behavior only when the environment observes certain restrictions on the production of inputs.

We want a problem specification to be an interface together with a set of behaviors. We therefore define a *schedule module* H to consist of two components, an action signature $sig(H)$, and a set $scheds(H)$ of *schedules*. Each schedule in $scheds(H)$ is a finite or infinite sequence of actions of H. Subject to the same restrictions as automata, schedule modules may be composed to form other schedule modules. The resulting signature is defined as for automata, and the schedules $scheds(H)$ is the set of sequences β of actions of H such that for every module H' in the composition, $\beta|H'$ is a schedule of H'.

It is often the case that an automaton behaves correctly only in the context of certain restrictions on its input. A useful notion for discussing such restrictions is that of a module 'preserving' a property of behaviors. A module *preserves* a property $\mathcal{P}$ iff the module is not the first to violate $\mathcal{P}$: as long as the environment only provides inputs such that the cumulative behavior satisfies $\mathcal{P}$, the module will only perform outputs such that the cumulative behavior satisfies $\mathcal{P}$. One can prove that a composition preserves a property by showing that each of the component automata preserves the property.

3 The Problem

In this section, we describe the architecture of the problem and then present a schedule module defining correctness for a multicast protocol.

3.1 The Architecture

Let $\mathcal{I} = \{1, \ldots, n\}$. Let $\mathcal{S}$ denote a universal set of text strings, and let $\mathcal{M}$ denote a universal set of messages, where both sets contain the empty sequence (ϵ). Let $u_i, i \in \mathcal{I}$, denote the n user processes engaged in the computation, and let $p_i, i \in \mathcal{I}$, denote n additional processes. Together, the p_i's are to solve the multicast problem, and each p_i is said to "work for" u_i. Each of the u_i's and p_i's is modelled as an automaton.

Each user u_i directly communicates by shared actions with the process p_i only. (One may think of u_i and p_i as running on the same processor.) The p_i's communicate with each other asynchronously via a network, also modelled as an automaton, that guarantees eventual one-time delivery of each message sent. Furthermore, we assume that all messages sent between each pair of processes are delivered in FIFO order.

The boundary between u_i and p_i is defined by several actions. To summarize the relationship between u_i and p_i at each point in an execution, we say that p_i is in a certain *region*, according to which of these actions have occurred. We will formalize this later. Figure 1 illustrates the actions shared by u_i and p_i, and by p_i and the network. Figure 2 illustrates possible region changes for p_i, and the actions that cause them.

Initially, p_i is in its "passive" region (P). We say that p_i enters its "trying" region (T) when user u_i issues a $try_i(S \subseteq \mathcal{I})$[1] action, indicating that u_i would like to send a multicast message to

[1] That is. $try_i(S)$, where $S \subseteq \mathcal{I}$.

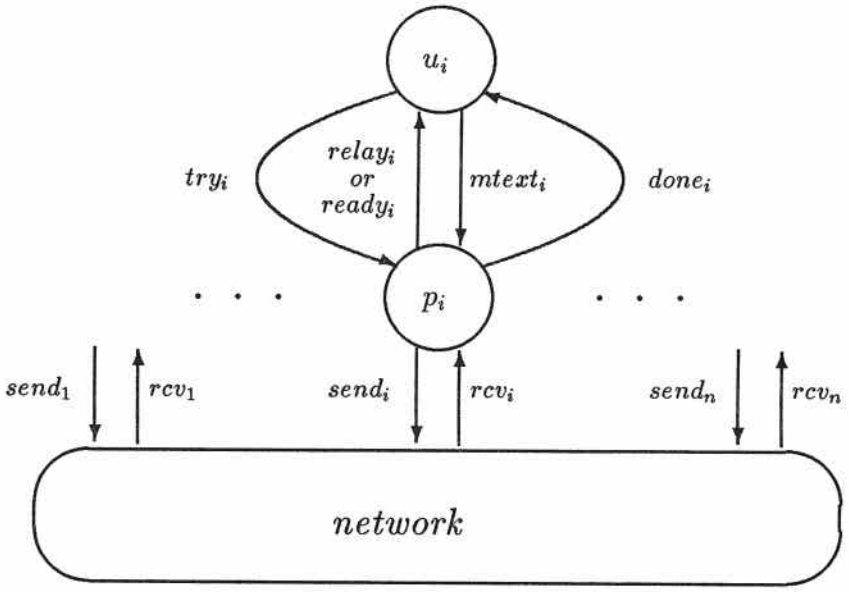

Figure 1: System Architecture. Arguments of actions are omitted.

processes named in the set S. When it is ready to perform a multicast on behalf of u_i, process p_i issues a $ready_i$ action and is said to enter its "ready" region (R). After receiving the $ready_i$ action as input, user u_i may issue a $mtext_i(m \in \mathcal{S})$ action, where the argument indicates the desired text of the multicast message. Upon receiving the $mtext_i$ action, p_i is said to enter its "bye" region (B), where it completes the multicast and returns to region P by issuing a $done_i$ action.

In addition to these actions, we have $relay_i(m \in \mathcal{S})$ actions, which are outputs of p_i and inputs to u_i. The purpose of these actions, which may occur while p_i is in P or T, is to forward multicast messages to u_i that were sent to p_i by some process p_j on behalf of user u_j. The argument m is the text of the multicast message. To correspond with this additional type of action, we have a "waiting" region (W), which is entered whenever p_i issues a $relay_i$ action while in T.[2] In W, p_i waits to see if u_i has "changed its mind" about the multicast after hearing the information contained in the $relay_i$ action. Either u_i still wishes to perform some multicast and issues a $try_i(S')$ action, or u_i decides not to do a multicast after all and issues an $mtext_i(\epsilon)$ action.

It might seem that one could eliminate region W and the $mtext(\epsilon)$ actions by having $relay_i$ actions take p_i to region P. However, this would make it difficult to express the liveness notion that u_i must eventually be allowed to perform a multicast, provided that it continually wants to do so. Region W is used to signify that u_i has a choice of continuing to try or "giving up."

3.2 Correctness

Note that the actions under the control of the protocol are exactly those actions that are the outputs of the p_i's. We only wish to require that the protocol is correct when its environment, namely the composition of the u_i's and the network, is well-behaved. We define schedule modules

[2] A $relay_i$ action from region P does not cause a region change.

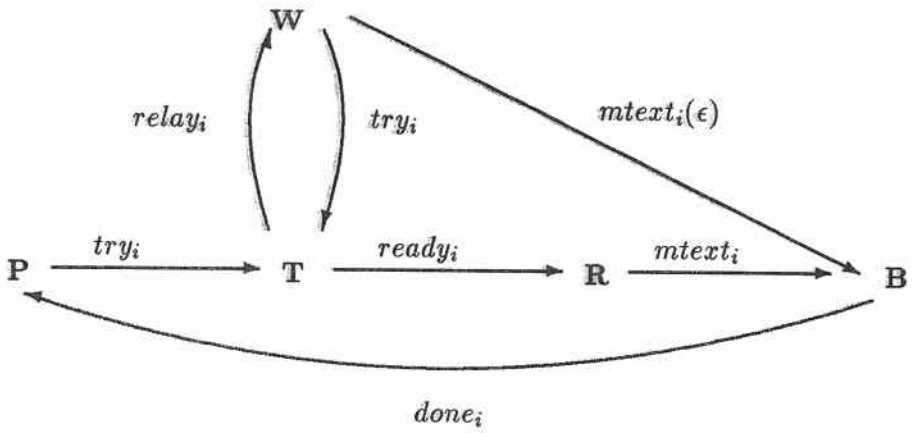

Figure 2: Region Changes for p_i. In region P, a $relay_i$ action does not cause a region change, and can be thought of as a self-loop.

to specify the allowable behaviors of each u_i and the network. Then we define a schedule module for the multicast protocol. We begin with the schedule modules for the u_i's.

Schedule Module U_i: We define the signature of U_i as follows:
$\text{in}(U_i) = \{relay_i(m \in \mathcal{S}), ready_i, done_i\}$
$\text{out}(U_i) = \{try_i(S \subseteq \mathcal{I}), mtext(m \in \mathcal{S})\}$

Before defining the set of schedules of U_i, we define a "region sequence" to capture the series of region changes in a schedule, and then state a *well-formedness* condition which makes use of this definition. Let the alphabet Σ = {P, T, R, B, W}. If L is a language, we let *prefixes*(L) denote the set of all prefixes of strings in L. Let α be an arbitrary sequence of actions. We define the *region of i after α*, denoted $r(i, \alpha)$, to be an element of Σ defined recursively as follows. If $\alpha|U_i$ is empty (ϵ), then $r(i, \alpha)$ = P. If $\alpha = \alpha'\pi$, then

$$r(i,\alpha) = \begin{cases} r(i,\alpha') & \text{if } \pi \notin U_i, \\ \text{T} & \text{if } \pi = try_i, \\ \text{R} & \text{if } \pi = ready_i, \\ \text{B} & \text{if } \pi = mtext_i, \\ \text{W} & \text{if } \pi = relay_i \wedge r(i,\alpha') \neq \text{P}, \\ \text{P} & \text{otherwise.} \end{cases}$$

Given an arbitrary action sequence α and an index $i \in \mathcal{I}$, let the *region sequence for i in α*, denoted *region-sequence*(i, α), be the string of characters over Σ constructed as follows. Let intermediate string σ be the concatenation of $r(i, \alpha')$ for each prefix of α in order, starting with $r(i, \epsilon)$ and ending with $r(i, \alpha)$. Then, to obtain *region-sequence*(i, α), remove from σ each character that is identical to its predecessor. Let α be an arbitrary sequence of actions. We say that α is *well-formed for i* iff

1. If $\alpha = \alpha'\pi$ with $\pi = mtext_i(m \neq \epsilon)$, then $r(i, \alpha')$ = R, and
2. *region-sequence*$(i, \alpha) \in$ prefixes(L), where the language L over Σ is defined by the regular expression P((TW)*TRBP + (TW)$^+$BP)*.

The language L reflects the fact that u_i is allowed to "change its mind" about sending a multicast message whenever it is relayed a message. We can now define the set of schedules for U_i.

Let α be a sequence of actions in sig(U_i). Then $\alpha \in$ scheds(U_i) iff:

1. U_i preserves well-formedness for i in α, and
2. region-sequence(i, α) does not end in W or R.

The first condition will be used in the safety proof, and the second in the liveness proof.

Schedule Module N: We now define a schedule module specifying the allowable behaviors of the network. The signature is as follows:

in(N) = $\{send(m \in \mathcal{M}, i, j \in \mathcal{I})\}$
out(N) = $\{rcv(m \in \mathcal{M}, i, j \in \mathcal{I})\}$

Network well-formedness: Let α be an arbitrary sequence of actions. Then α is said to be *network well-formed* iff

1. For each $send(m, i, j)$ event in α, there exists at most one later $rcv(m, i, j)$ event in α. (We call this event the *rcv corresponding to* $send(m, i, j)$.)
2. For each $rcv(m, i, j)$ event in α, there exists an earlier $send(m, i, j)$ event in α (called the *corresponding send*).
3. If event $rcv(m, i, j)$ occurs in α before event $rcv(m', i, j)$, then their corresponding events $send(m, i, j)$ and $send(m', i, j)$ occur in the same order.

These conditions say that each message is delivered at most once, no spurious messages are delivered, and that messages between any pair of processes are delivered in the order sent.

Given the network well-formedness definition, we can define the set of schedules of N. Let α be an sequence of actions of N. Then $\alpha \in$ scheds(N) iff

1. α is network well-formed, and
2. for each $send(m, i, j)$ event in α, there exists a $rcv(m, i, j)$ event later in α.

The second condition states that every message sent is eventually delivered.

Schedule Module M: We can now rephrase the correctness conditions informally stated in Section 1 in terms of the actions at the boundaries of the user processes. We do this with a schedule module M, which defines the multicast problem. Let sig(U) = $\cup_{i \in \mathcal{I}}$ sig(U_i).[3] We define the signature of M as follows:

in(M) = out(U) $\cup$ out(N)
out(M) = in(U) $\cup$ in(N)

Let α be a sequence of actions of sig(M). Then $\alpha \in$ scheds(M) iff:

1. $\forall i \in \mathcal{I}$, M preserves well-formedness for i in α.

[3] That is, component-wise union: $in(U) = \cup_{i \in \mathcal{I}} in(U_i)$, etc.

2. If α is well-formed for every $i \in \mathcal{I}$ and α is network well-formed, then $\forall j \in \mathcal{I}$ and $\forall m, m' \in \mathcal{S}$,

   (a) If $relay_i(m)$, $relay_i(m')$, $relay_j(m)$, and $relay_j(m')$ occur in α, and if $relay_i(m)$ precedes $relay_i(m')$, then $relay_j(m)$ precedes $relay_j(m')$.

   (b) If $mtext_i(m \neq \epsilon)$ occurs in α, then no $relay_i(m' \neq m)$ occurs between $mtext_i(m)$ and $relay_i(m)$.

3. If $\alpha|N \in \text{scheds}(N)$ and $\forall i \in \mathcal{I}$, $\alpha|U_i \in \text{scheds}(U_i)$, then the following hold:

   (a) If a try_i occurs in α and each $relay_i$ thereafter is immediately followed by a try_i in $\alpha|U_i$, then a $ready_i$ occurs later in α.

   (b) If an $mtext_i(m \neq \epsilon)$ occurs in α and $try_i(S)$ is the last preceding try_i action in α, then a $relay_j(m)$ occurs later in α for each $j \in S$.

Items (1) and (2) are the required *safety* properties, and item (3) is the required *liveness* property. A multicast protocol is *correct* iff it solves M.

4 The Algorithm

This section presents the multicast protocol. We present the algorithm by giving an explicit I/O automaton for each p_i, $i \in \mathcal{I}$. We show in Section 5 that the composition of the p_i's solves the schedule module M and is therefore a correct protocol.

The algorithm is based on logical time. We define a *logical time* to be an (integer, process-id) pair drawn from $\mathcal{T} = (\{1, 2, \ldots\} \cup \infty) \times \mathcal{I}$, and we let logical times be ordered lexicographically. Essentially, each process p_i maintains a logical time clock, and each multicast is assigned a unique logical time. The process p_i relays all multicast messages destined for u_i in logical time order.[4]

4.1 An Informal Overview

We begin with a discussion of the main ideas of the algorithm, which is followed by the formal automaton definition. Unless otherwise noted, the word "process" will refer to one of the processes p_i, $i \in \mathcal{I}$. Also, we use the words "time" and "logical time" interchangeably.

On receiving a $try_i(S)$ input, p_i remembers S as its *try-set*. In region T, p_i tries to set up a multicast for u_i to processes named in *try-set*, and it may request permission to do so from processes $p_j, j \in \mathcal{I}$ by sending "req-promise" messages. Permission is granted from a process p_j to p_i in the form of a "promise" message with an associated logical time t. The promise means that p_j will not perform any multicasts with a time greater than t until p_i either explicitly relinquishes the promise (by sending a "bye" message to p_j) or advances the promise (by sending an "adv-promise" message with the later time). One may think of this promise as a roadblock that p_i erects in u_j's computation at some future logical time. The process p_j doesn't allow u_j's computation to advance past that time until the roadblock is removed or moved forward by p_i. Processes are guaranteed of eventually getting a promise in response to every request.

Every multicast performed by p_i has an associated logical time (its *btime*), which is assigned by p_i to be at least as large as both p_i's *clock* and the maximum logical time among all promises held by p_i. Processes communicate their multicasts in "bye" messages, which have two arguments: the text of the multicast and its assigned logical time. (Upon receipt of a "bye" message, a process p_j keeps the (text, time) pair in a *pending* set until the message is relayed to u_j. It is safe to relay a

[4] We never assign a time of ∞ to a multicast message; it is used only as an upper bound.

message m when it is the message with the smallest time t in *pending* and all promises granted by p_j with times lower than t have either been relinquished or advanced past t.)

After receiving a promise from each process p_j to whom a request was made, p_i is ready perform a multicast with $btime = t$ provided that (1) its *pending* set is empty, and (2) all promises p_i has granted with times lower than t have either been relinquished or advanced past t. The second condition is present to ensure that u_i receives no new messages with logical times less than t after p_i decides to send its multicast.

Finally, we explain those conditions under which p_i may advance promises. Suppose that p_i has received promises from all processes in its *try-set*, but has determined that it is not yet ready to perform a multicast to relinquish those promises. In order to not unnecessarily block the computation of those processes from which p_i has received promises, it may send "adv-promise" messages to those processes, informing them of the earliest possible time that p_i might actually use as its *btime*. Without these messages, it would be possible for the computation to deadlock.[5]

4.2 The Detailed Algorithm

We now present the formal algorithm description. The state of p_i has several components: A variable *region* $\in$ {P,T,W,R,B} is initially set to P. The variables *try-set*, *requested*, and *requests* are subsets of $\mathcal{I}$, initially empty. The variable *text* $\in \mathcal{S}$ is initially the empty string (ϵ). Two arrays of logical times indexed by $\mathcal{I}$ are kept: *promises-to* and *permission-from*. All entries of these arrays are initially greater than any possible logical time (∞, n). Two additional logical time variables, *clock* and *btime*, are initially $(0, i)$. Finally, the variable *pending* is an initially empty set of (text $\in \mathcal{S}$, time $\in \mathcal{T}$) pairs.

We let min(*promises-to*) denote the smallest time among the entries in that array. Similarly, we let max(*permission-from*) denote the largest time less than (∞, n) among the entries in that array; if all entries in that array are (∞, n), then max(*permission-from*) $= (\infty, n)$. Finally, we let min(*pending*) and max(*pending*) denote the pairs in the set having the least and greatest logical times, respectively; if *pending* is empty, then both values are $(\epsilon, (0, i))$.

In addition to the above variables, the algorithm description refers to a pseudo-variable, *try-time*, defined according to p_i's state components as follows: *try-time* is the smallest logical time having process-id i such that

$$\textit{try-time} \geq \max(\textit{clock}, \textit{btime}, \max(\textit{permission-from})).$$

Automaton p_i has the following signature.

Input actions:	$try_i(S \subseteq \mathcal{I})$
	$mtext_i(m \in \mathcal{S})$
	$rcv(m \in \mathcal{M}, j \in \mathcal{I}, i)$
Output actions:	$relay_i(m \in \mathcal{S})$
	$ready_i$
	$done_i$
	$send(m \in \mathcal{M}, i, j \in \mathcal{I})$

The transition relation for p_i is shown in Figure 3. "P" and "E" denote precondition and effect, respectively. An action is enabled in exactly those states in which the precondition is satisfied. If

[5] Consider a situation in which p_i and p_j are trying to send multicasts such that each is in the other's *try-set*. Suppose that all promises received by p_i are smaller than some promise received by p_j. If p_i has granted p_j a promise smaller than p_i's own *try-time*, neither can perform a multicast before the other.

Input Actions:

- $try_i(S)$
  E: $try\text{-}set = S \cup \{i\}$
  $region = \mathrm{T}$
- $mtext_i(m)$
  E: $text = m$
  if $m = \epsilon$ **then** $try\text{-}set = \emptyset$
  $btime = try\text{-}time$
  $region = \mathrm{B}$
- rcv(req-promise,$j \in \mathcal{I}, i$)
  E: $requests = requests \cup \{j\}$
- rcv(bye($m \in \mathcal{S}, t \in \mathcal{T}$),$j \in \mathcal{I}, i$)
  E: $promises\text{-}to[j] = (\infty, n)$
  if $m \neq \epsilon$ **then**
  $pending = pending \cup \{(m,t)\}$
- rcv(adv-promise($t \in \mathcal{T}$),$j \in \mathcal{I}, i$)
  E: $promises\text{-}to[j] = t$
- rcv(promise($t \in \mathcal{T}$),$j \in \mathcal{I}, i$)
  E: $permission\text{-}from[j] = t$

Output Actions:

- $relay_i(m)$
  P: $region \in \{\mathrm{P},\mathrm{T}\}$
  $(m,t) = \min(pending)$
  $t < \min(promises\text{-}to)$
  E: $pending = pending \setminus \{(m,t)\}$
  $clock = t$
  if $region = \mathrm{T}$ **then** $region = \mathrm{W}$
- $ready_i$
  P: $region = \mathrm{T}$
  $pending = \emptyset$
  $\min(promises\text{-}to) \geq try\text{-}time$
  $\forall j \in requested$,
  $permission\text{-}from[j] < (\infty, n)$
  E: $region = \mathrm{R}$
- $done_i$
  P: $region = \mathrm{B}$
  $requested = \emptyset$
  E: $region = \mathrm{P}$
- $send$(promise($t \in \mathcal{T}$),$i, j \in \mathcal{I}$)
  P: $j \in requests$
  $t > \max(try\text{-}time, \max(pending).\mathrm{time})$
  E: $requests = requests \setminus \{j\}$
  $promises\text{-}to[j] = t$
- $send$(req-promise,$i, j \in \mathcal{I}$)
  P: $region \in \{\mathrm{T},\mathrm{W}\}$
  $j \notin requested$
  E: $requested = requested \cup \{j\}$
- $send$(adv-promise($t \in \mathcal{T}$),$i, j \in \mathcal{I}$)
  P: $region \in \{\mathrm{T},\mathrm{W}\}$
  $\forall k \in try\text{-}set$,
  $permission\text{-}from[k] < (\infty, n)$
  $permission\text{-}from[j] < try\text{-}time$
  $t = try\text{-}time$
  E: $permission\text{-}from[j] = try\text{-}time$
- $send$(bye($m \in \mathcal{S}, t \in \mathcal{T}$),$i, j \in \mathcal{I}$)
  P: $region = \mathrm{B}$
  $permission\text{-}from[j] < (\infty, n)$
  $t = btime$
  if $(j \in try\text{-}set)$ **then**
  $m = text$
  else $m = \epsilon$
  E: $requested = requested \setminus \{j\}$
  $permission\text{-}from[j] = (\infty, n)$.

Figure 3: Transition relation for p_i.

an action has no precondition, it is always enabled. When an action occurs, p_i's state is modified according to the assignments in the effects clause. If a state component is not mentioned in the effect clause, it is left unchanged by the action.

The equivalence classes of part(p_i) are as follows. The actions $relay_i$, $ready_i$, and $done_i$ are together in one class. And for each $j \in \mathcal{I}$, there exist four classes containing the sets of actions *send*(promise($t \in \mathcal{T}$),i,j), *send*(req-promise,i,j), *send*(adv-promise($t \in \mathcal{T}$),i,j), and *send*(bye($m \in \mathcal{S}, t \in \mathcal{T}$),$i,j$).

5 Proof of Correctness

Consider module P, the composition of all automata $p_i, i \in \mathcal{I}$. In this section, we show that module P solves schedule module M, which implies the correctness of the logically synchronous multicast protocol. The correctness proof follows the definition of schedule module M. Clearly, sig(P) = sig(M). To show that P solves M, we need to show that all fair behaviors of P satisfy the safety conditions (1 and 2) and the liveness condition (3). We prove these in order. To distinguish the state components of the different automata in P, we use subscripts. For example, $region_i$ is the *region* variable in the local state of automaton p_i.

5.1 Safety Proof

We start with Condition (1), that P preserves well-formedness for all $i \in \mathcal{I}$. If ω and ω' are strings over a common alphabet, we let $\omega \circ \omega'$ denote the concatenation of the two strings. The following lemma states some useful properties of language L (from the definition of well-formedness for i).

Lemma 1: For all strings $\omega \in$ prefixes(L), the following properties hold:

1. If ω ends in T, then ωoR $\in$ prefixes(L).
2. If ω ends in B, then ωoP $\in$ prefixes(L).
3. If ω ends in T, then ωoW $\in$ prefixes(L).

Proof: By inspection of the definition of L. ∎

The next lemma relates the state of p_i after an execution α to the definition of $r(i,\alpha)$.

Lemma 2: Let α be an execution of P ending in state s. Then for all $i \in \mathcal{I}$, $s.region_i = r(i,\alpha)$.

Proof: By induction on the length of α.

Base case: If α is of length 1 (just an initial state), then $r(i,\alpha)$ = P by definition for all $i \in \mathcal{I}$, since the schedule of $\alpha|U_i$ is the empty sequence. In every initial state, $region_i$ = P for all $i \in \mathcal{I}$, so the lemma holds.

Induction: Let $\alpha = \alpha'\pi s$, where the lemma holds for α' ending in state s'. For all i, if $\pi \notin$ sig(U_i), then $s.region_i = s'.region_i$. This satisfies the lemma, since $r(i,\alpha) = r(i,\alpha')$ by definition if $\pi \notin$ sig(U_i).

Now, if $\pi \in$ sig(U_i), then there are five cases: If π is a try_i action, then $s.region_i$ = T by the effects clause of try_i actions. By definition, if π is a try_i action, then $r(i,\alpha)$ = T, so the lemma holds. The next three cases, $ready_i$, $mtext_i$, and $done_i$, are argued similarly. The final case is $relay_i$, whose effects clause states that $s.region_i$ = W if $s'.region_i$ = T, and that $s.region_i = s'.region_i$ otherwise. Moreover, if $s'.region_i \neq$ T, then $s'.region_i$ = P, since $relay_i$ is enabled only when $region_i \in$ {P,T}. Therefore, $s.region_i = r(i,\alpha)$. ∎

We are now ready to prove that module P satisfies Condition (1) of schedule module M.

Theorem 3: Module P preserves well-formedness for i, for all $i \in \mathcal{I}$.

Proof: Consider execution $\alpha = \alpha' \pi s$ of P, where α' is well-formed for all $i \in \mathcal{I}$ and ends in state s'. Since $mtext_i$ is not an output action of P, we need not consider Part 1 of the well-formedness definition. Similarly, for Part 2 we need only consider cases where π is an output of P. We know by Lemma 2 that $s.region_i = r(i, \alpha)$. Also, we know that region-sequence$(i, \alpha') \in$ prefixes(L), since α' is well-formed for i. There are three cases for π an output of P.

1. $\pi = ready_i$: This is only enabled if $s'.region_i = $ T. Therefore region-sequence(i, α') ends in T. So, by Condition 1 of Lemma 1, region-sequence$(i, \alpha') \circ$ R $\in$ prefixes(L), and this is precisely region-sequence(i, α).

2. $\pi = done_i$: This is only enabled if $s'.region_i = $ B. Therefore region-sequence(i, α') ends in B. So, by Condition 2 of Lemma 1, region-sequence$(i, \alpha') \circ$ P $\in$ prefixes(L), and this is precisely region-sequence(i, α).

3. $\pi = relay_i$: This is only enabled if $s'.region_i = $ P or T. If $s'.region_i = $ P, then region-sequence(i, α) = region-sequence(i, α'), which is in prefixes(L). If $s'.region_i = $ T, then region-sequence(i, α') ends in T. So, by Condition 3 of Lemma 1, region-sequence$(i, \alpha') \circ$ W $\in$ prefixes(L), and this is precisely region-sequence(i, α).

■

Given that module P preserves well-formedness for all $i \in \mathcal{I}$, the following definition is useful for restricting attention to the executions of P in which the environment is well-behaved. (Note that because no *rcv* action is an output of P, it is not possible for P to violate network well-formedness.)

Let α be an execution of P. We say that α is *admissible* iff α is well-formed for every $i \in \mathcal{I}$ and α is network well-formed.

To show that module P satisfies Condition (2a), that all multicast messages are delivered in the same relative order, we use the following sequence of lemmas. The first lemma states some useful facts about the ordering on events in executions of P.

Lemma 4: Let α be an admissible execution of P. Let α' be a subexecution of P between two successive $done_i$ actions (or between the beginning of α and the first $done_i$ action). Then if α' contains *any one* of the following five actions, then it contains *exactly one* of each of them such that they occur in the following order: *send*(req-promise,i, j), *rcv*(req-promise,i, j), *send*(promise(t),j, i), *rcv*(promise(t),j, i), and *send*(bye(m, t''),i, j). Furthermore, if a *send*(adv-promise(t'),i, j) occurs in α', then it occurs between the last two actions.

Proof: The proof is by induction, assuming that the conditions hold for the prefix of α up to the beginning of α'.

First we show that no two *send*(req-promise,i, j) events can occur. The action π_1 = *send*(req-promise,i, j) is only enabled when $region_i = $ T and $j \notin requested_i$. When the action occurs, it results in $j \in requested_i$. The set $requested_i$ can only be decreased when $region_i = $ B. Therefore, another action *send*(req-promise,i, j) cannot occur after π_1 until p_i passes through some state in which $region_i = $ B and then reaches a state in which $region_i = $ T. By well-formedness for i, this cannot happen without an intervening $done_i$.

Next, we show that if π_1 = *send*(req-promise,i, j) occurs in α, then the next $done_i$ event after π_1 must be preceded by π_5 = *send*(bye(m, t''),i, j). The action π_1 has as an effect that $j \in requested_i$,

and $done_i$ has as a precondition that $requested_i$ is empty. Therefore, since π_5 is the only action that can remove j from $requested_i$, it must occur between π_1 and π_5.

Now we show that each event in the sequence must occur in order for the next to occur. By the induction hypothesis, all *send*(req-promise,i,j) actions have their corresponding receives before the start of α'. Therefore, by network well-formedness, $\pi_2 = rcv$(req-promise,i,j) cannot occur before π_1, and only one π_2 action occurs. Action $\pi_3 = send$(promise(t),j,i) is only enabled when $i \in requests_j$, and the event results in i's removal from that set. Since π_2 is the only action that can cause $i \in requests_j$, it must precede π_3. Again, by network well-formedness and the induction hypothesis, we know that π_3 must precede $\pi_4 = rcv$(promise(t),j,i). The action $\pi_5 = send$(bye(m,t''),i,j) has as a precondition that $permission\text{-}from_i[j] \; ¡ \; (\infty, n)$. Since π_5 has as an effect that $permission\text{-}from_i[j] = (\infty, n)$, and since π_4 is the only action that can cause $permission\text{-}from_i[j] < (\infty, n)$, we know by the induction hypothesis that $permission\text{-}from_i[j] = (\infty, n)$ at the beginning of α'. Therefore, π_4 must precede π_5.

Since *send*(adv-promise(t'),i,j) has as a precondition that $permission\text{-}from_i[j] < (\infty, n)$, we know that it cannot occur before π_4 or after π_5. ■

The above lemma is used to show the existence or nonexistence of certain events in a portion of an execution.

The following lemma states some important invariants on the state of P. The fifth invariant, which states that the minimum time in the pending set of a process p_i is always larger than the clock of that process, is a key piece of the safety proof. Informally, it tells us that no multicast message arrives "too late".

Lemma 5: Let α be an admissible execution of P. Then for all $i, j \in \mathcal{I}$, the following properties hold for all states s in α.

1. $i \in s.requests_j \Rightarrow s.promises\text{-}to_j[i] = (\infty, n)$
2. $s.promises\text{-}to_j[i] \leq s.permission\text{-}from_i[j]$
3. $s.clock_j < s.promises\text{-}to_j[i]$
4. $(s.region_i = \mathrm{B} \wedge j \in s.try\text{-}set_i \cap s.requested_i) \Rightarrow s.permission\text{-}from_i[j] \leq s.btime_i$
5. $s.pending_j \neq \emptyset \Rightarrow s.clock_j < min(s.pending_j).\mathrm{time}$

Proof: Property (1) is proved by induction on the length of α. If s is an initial state, then for all $i, j \in \mathcal{I}$, $i \notin s.requests_j$, so the statement holds vacuously. The only action which can add i to $requests_j$ is a rcv(req-promise,i,j). So, for the induction step, let $\alpha = \alpha'\pi s$, where $\pi = rcv$(req-promise,i,j) and Property (1) holds for α'. Suppose (for contradiction) that $s.promises\text{-}to_j[i] < (\infty, n)$. This can only be true if there exists some π', either a *send*(promise(t),j,i) or a rcv(adv-promise(t),i,j), in α' such that no rcv(bye(m,t'),i,j) occurs between π' and π. However, by Lemma 4, every *send*(promise(t),j,i) or *send*(adv-promise(t),i,j) must be followed by a *send*(bye(m,t'),i,j) before the next *send*(req-promise,i,j) occurs. And so by network well-formedness, rcv(bye(m,t'),i,j) must occur between π' and π, giving us a contradiction.

Property (2) is also proved by induction on the length of α. The base case, α only a start state, holds since $permission\text{-}from_i[j] = promises\text{-}to_j[i] = (\infty, n)$ for all $i, j \in \mathcal{I}$. Let $\alpha = \alpha' s' \pi s$ be an execution of P, where the property holds in state s'. Now, consider those four actions π that can potentially increase $promises\text{-}to_j[i]$ or decrease $permission\text{-}from_i[j]$:

1. If π = *send*(promise(t),j,i), then by Property (1) and the preconditions on π, s'.*promises-to*$_j[i] = (\infty, n)$. Therefore, *promises-to*$_j[i]$ is not increased by π.

2. If π = *rcv*(promise(t),j,i), then s.*permission-from*$_i[j] = t$. By network well-formedness, π' = *send*(promise(t),j,i) must occur earlier in α'. The only possible events that could occur between π' and π to make s.*promises-to*$_j[i] \neq t$ are *rcv*(adv-promise(t'),i,j) or *rcv*(bye(m,t'),i,j). By Lemma 4, we know that π' = *send*(promise(t),j,i) must occur before π such that no *send*(adv-promise(t'),i,j) or *send*(bye(m,t'),i,j) occurs between π' and π. Furthermore, by the same lemma, we know that a *rcv*(req-promise,i,j) occurs before π' and after any *send*(adv-promise(t'),i,j) or *send*(bye(m,t'),i,j). Hence, by network well-formedness, no *rcv*(adv-promise(t'),i,j) or *rcv*(bye(m,t'),i,j) occurs between π' and π.

3. If π = *rcv*(adv-promise(t'),i,j), then s.*promises-to*$_j[i] = t'$. By Lemma 4 and network well-formedness, the corresponding *send*(adv-promise(t'),i,j) must follow a π' = *send*(promise(t),j,i) such that no *rcv*(bye(m,t'''),i,j) occurs between them. By the preconditions of *send*(adv-promise(t'),i,j), $t' > t$, and that action results in *permission-from*$_i[j] = t'$. Furthermore, any other *send*(adv-promise(t''),i,j) occuring in α' after *send*(adv-promise(t'),i,j) must have $t'' > t'$. Therefore, the property holds.

4. If π = *rcv*(bye(m,t),i,j), then s.*promises-to*$_j[i] = (\infty, n)$. By network well-formedness, π must be preceded by π' = *send*(bye(m,t),i,j), resulting in *permission-from*$_i[j] = (\infty, n)$. The only action that could can decrease *permission-from*$_i[j]$ is a *rcv*(promise(t'),j,i). But by Lemma 4, any *rcv*(promise(t'),j,i) occuring between π' and π must be preceded in that interval by a *send*(req-promise,i,j) and a *rcv*(req-promise,i,j). And this violates well-formedness, so no *rcv*(promise(t'),j,i) occurs between π' and π. Therefore s.*permission-from*$_i[j] = (\infty, n)$.

Property (3) is also proved by induction on the length of α. The base case, α a start state, holds since *clock*$_j = (0, j)$ and *promises-to*$_j[i] = (\infty, n)$ for all $i \in \mathcal{I}$. Now, consider those actions that can potentially increase *clock*$_j$ or decrease *promises-to*$_j[i]$. These are *relay*$_j$, *send*(promise(t),j,i), and *rcv*(adv-promise(t),i,j). By definition, the action *relay*$_j$ sets clock to a value t, such that $\forall i \in \mathcal{I}$, *promises-to*$_j[i] > t$. The action *send*(promise(t),j,i) sets *promises-to*$_j[i] = t$ and is enabled only if $t >$ *try-time*, which is at least *clock*$_j$ by definition. Finally, the action *rcv*(adv-promise(t),i,j) sets *promises-to*$_j[i] = t$. To show that $t >$ *clock*$_j$, we note that *send*(adv-promise(t),i,j) is enabled at p_i only if *permission-from*$_i[j] < t$. Therefore, by Property (2), $t >$ *promises-to*$_j[i]$ when *send*(adv-promise(t),i,j) occurs. And therefore, $t >$ *promises-to*$_j[i]$ when *rcv*(adv-promise(t),i,j) occurs, since Lemma 4 and network well-formedness tell us that neither a *rcv*(bye(m,t'),i,j) nor a *send*(promise(t'),j,i) action can occur between *send*(adv-promise(t),i,j) and *rcv*(adv-promise(t),i,j).

Property (4) is proved by a simple induction on the length of α. We observe that whenever p_i enters region B (an *mtext*$_i$ action), *try-time* and *btime* are made equal and that *btime* remains unchanged until after p_i is no longer in region B. We also observe that by well-formedness for i, no *try*$_i$ actions can occur from region B, so *try-set*$_i$ is also fixed.

If p_i does not enter B from region R, then the action to enter B must be *mtext*$_i(\epsilon)$, by well-formedness for i. Therefore, by the effects clause of that action, *try-set*$_i = \emptyset$, so the property holds vacuously. If p_i does enter B from region R, then by the preconditions on *ready*$_i$, promises are held by p_i for all in *try-set*$_i$. And by the effects of *ready*$_i$ and the definition of *try-time*, *btime* is set to a value at least as great as any of those promises. Therefore, the property holds on entering B.

By Lemma 4, no new promises from members of *try-set* are received by p_i while in B, since promises have already been received. Therefore, it suffices to show that for all $j \in$ *try-set*, if *permission-from*$_i[j]$ is increased, then j is removed from *requested* until the next *done*$_i$. Since

send(adv-promise(t),i,j) actions are not enabled from B, we only need consider *send*(bye(m,t),i,j). However, this action removes j from *requested*. Since *send*(req-promise,i,j) is not enabled in B, j cannot be replaced in *requested* before the next $done_i$.

Property (5) can be shown by induction using the above invariants. Clearly, the property holds in the initial state. Let $\alpha = \alpha' s' \pi s$ be an execution of P, where the property holds in state s'. The only action that can change $clock_j$ is a $relay_j$, which removes the element from $pending_j$ having the lowest time, and sets $clock_j$ to that time. Therefore, by the induction hypothesis, the property holds.

The action $\pi = rcv(\text{bye}(m \neq \epsilon, t),i,j)$, for some $i \in \mathcal{I}$, is the only action that can add elements to $pending_j$. Let s'' be the state from which the corresponding *send*(bye(m,t),i,j) occurs. Since $m \neq \epsilon$ implies that $j \in s''.try\text{-}set_i$, we know from Property (4) that $s''.permission\text{-}from_i[j] \leq t = s''.btime_i$. Therefore, by Property (2), $s''.promises\text{-}to_j[i] \leq t$ By Lemma 4 and network well-formedness, we know that no *send*(promise(t'),j,i) or rcv(adv-promise(t'),i,j) action can occur between s'' and s' to could cause $promises\text{-}to_j[i]$ to increase past t. Therefore $s'.promises\text{-}to_j[i] \leq t$. So, by Property (3), $s'.\text{clock}_j < t$. So, when π occurs, (m,t) is added to $pending_j$ and the Property (5) holds in state s. ■

The following lemmas state that the state components *clock* and *btime* are nondecreasing.

Lemma 6: Let α be an admissible execution of P. Then for all $i \in \mathcal{I}$, if state s' precedes state s in α, then $s'.clock_i \leq s.clock_i$.

Proof: Consider the actions $relay_i$, which are the only actions in which $clock_i$ can be modified. Whenever a $relay_i$ action is enabled, $pending_i$ is nonempty. By definition, a $relay_i$ action results in $clock_i$ being set to the minimum logical time in $pending_i$. By Property (5) of Lemma 5, $clock_i$ is less than the mininum logical time in $pending_i$, provided $pending_i$ is nonempty. Therefore, whenever $clock_i$ is modified, its value is increased. ■

Lemma 7: Let α be an admissible execution of P. Then for all $i \in \mathcal{I}$, if state s' precedes state s in α, then $s'.btime_i \leq s.btime_i$.

Proof: Consider the actions $mtext_i$, which are the only actions in which $btime_i$ can be modified. These actions set $btime_i$ to the value of $try\text{-}time_i$, which is no smaller than $btime_i$ by definition. ■

The following lemma states that each multicast message is assigned a unique logical time.

Lemma 8: Let α be an admissible execution of P. Let *send*(bye(m,t),i,j) and *send*(bye(m',t'),i',j') occur in α. If $m \neq m'$ or $i \neq i'$, then $t \neq t'$.

Proof: If $i \neq i'$, then clearly $t \neq t'$ because they differ in the process-id. If $m \neq m'$ and $i = i'$ then m and m' are the text of different multicasts by u_i. Without loss of generality, suppose $mtext_i(m)$ precedes $mtext_i(m')$. Then, by Lemma 7 and the definition of *try-time*, the *btime* assigned to m is greater than that assigned to m'. Therefore, $t > t'$. ■

The following two theorems prove that executions of P satisfy conditions (2a) and (2b) of schedule module M.

Theorem 9: Let α be an admissible execution of P. Then $\forall j \in \mathcal{I}$ and $\forall m, m' \in \mathcal{S}$, if $relay_i(m)$, $relay_i(m')$, $relay_j(m)$, and $relay_j(m')$ occur in α, and if $relay_i(m)$ precedes $relay_i(m')$, then $relay_j(m)$ precedes $relay_j(m')$.

Proof: From Lemma 8, we know that m and m' have different *btimes*, say t and t'. Without loss of generality, let $t' > t$. Suppose that for some $j \in \mathcal{I}$, $relay_j(m')$ precedes $relay_j(m)$. By the definition of the $relay_j$ action, the message with the smallest logical time in $pending_j$ is delivered.

Therefore, in the state from which $relay_j(m')$ occurs, $(m,t) \notin pending_j$. The effects clause states that $relay_j(m')$ results in $clock_j = t'$. So, by Lemma 6, $clock_j \geq t'$ at all later states in α. In order for $relay_j(m)$ to be enabled from one of these later states, it must be the case that eventually $(m,t) \in pending_j$. However, since $t < t'$ and $clock_j \geq t'$, this contradicts Property (5) of Lemma 5 that $clock_j < min(pending_j).$time. Therefore, $relay_j(m)$ precedes $relay_j(m')$. ■

Theorem 10: Let α be an admissible execution of P. Then $\forall j \in \mathcal{I}$ and $\forall m, m' \in \mathcal{S}$, if $mtext_i(m)$ occurs in α, then no $relay_i(m')$ occurs between $mtext_i(m)$ and $relay_i(m)$.

Proof: Consider the state s from which $mtext_i(m \neq \epsilon)$ occurs, and let α' be the prefix of α ending in state s. We know, from well-formedness for i, that $r(i,\alpha') = \mathrm{R}$. Consider the last action $ready_i$ occurring in α', and let s' be the resulting state. (We know such an action must occur, since this is the only action that can result in region R.) We know, again by well-formedness for i, that $region_i = \mathrm{R}$ at all states between s' and s.

By the preconditions of $ready_i$, it is true that $s'.permission\text{-}from_i[j] < (\infty, n)$, for all $j \in s'.requested_i$. Therefore, since no $send$(req-promise,i,j) actions are enabled from R, we know by Lemma 4 that no promises are received by p_i between s' and s. Therefore, since $btime_i$ cannot change in R and $clock_i$ cannot change until a $relay_i$ occurs, we can conclude by the definition of *try-time* that $try\text{-}time_i$ is fixed between s' and s unless a $relay_i$ occurs. Again by the preconditions on $ready_i$, $s'.\mathrm{pending}_i = \emptyset$ and $s'.promises\text{-}to_i[j] \geq s'.try\text{-}time$, for all $j \in \mathcal{I}$. Also, any $send$(promise(t'),i,j) must have $t' > try\text{-}time$, so by Properties (2) and (4) of Lemma 5, any rcv(bye(m',t''),j,i) must have $t'' > s'.try\text{-}time_i$. Therefore, no $relay_i$ can occur and *try-time* is fixed between s' and s.

Now, when $mtext_i(m)$ occurs after state s, the *btime* for m is set to $t = s'.try\text{-}time > s.promises\text{-}to_i[j], \forall i \in \mathcal{I}$. So, by Lemma 7, all later $send$(promise(t'),i,j) must have $t' > t$. Again, by Properties (2) and (4) of Lemma 5, any rcv(bye(m',t''),j,i) after s must have $t'' > t$. Since any m' with a *btime* less than t must have been relayed to u_i prior to s' (by precondition on $ready_i$), we know that no $relay_i(m')$ occurs between $mtext(m)$ and $relay_i(m)$. ■

5.2 Liveness Proof

We now proceed with the liveness proof. The following definition will be convenient for limiting ourselves to the discussion of executions in which the environment is "well-behaved". Let α be an fair execution of P. We say that α is *well-behaved* iff $\alpha|U_i \in \mathrm{scheds}(U_i)$ for all $i \in \mathcal{I}$ and $\alpha|N \in \mathrm{scheds}(N)$. Note that every well-behaved execution is an admissible execution, by the definitions of U_i and N, and the fact that P preserves well-formedness for all $i \in \mathcal{I}$. The following lemma states that if a promise is requested, then eventually it is granted.

Lemma 11: Let α be a well-behaved execution of P. If event $\pi = send$(req-promise,i,j) occurs in α then a later rcv(promise(t),j,i) occurs in α.

Proof: By the definition of scheds(N), a $\pi' = rcv$(req-promise,i,j) occurs in α after π. By the transition relation for p_j, $requests_j[i] < (\infty, n)$ in the state after π' (specifically, its value is t), and only a $send$(promise(t),j,i) action can cause $requests_j[i] = (\infty, n)$. Therefore, $send$(promise(t),j,i) is enabled in all states after π' until it occurs. By the definition of scheds(N), a corresponding rcv(promise(t),j,i) occurs later in α. ■

The following lemma states a property of executions of P that is useful in proving Lemma 13.

Lemma 12: Let α be a well-behaved execution of P. If a try_i action occurs in α, then either a $mtext_i(\epsilon)$ action occurs later in α, or there must exist a point later in α after which the following

condition holds for all states s up to the next $ready_i$ action: $\forall j \in s.requested$, $s.permission\text{-}from_i[j] < (\infty, n)$.

Proof: Suppose not. If no $mtext_i(\epsilon)$ action occurs in α after the try_i event, then by well-formedness for i, $s'.region_i \in \{\text{T,W}\}$ in all states s' after the try_i until a $ready_i$ occurs. However, by our supposition, no state after the try_i satisfies the conditions for state s in the statement of the lemma, so $ready_i$ is not enabled in any state after the try_i. So, $s'.region_i \in \{\text{T,W}\}$ in all states s' after the try_i.

Therefore, for all such states s', for all $j \in \mathcal{I}$, either $j \notin s'.requested_i$, and $send(\text{req-promise},i,j)$ is enabled or $j \in s'.requested_i$ and $send(\text{req-promise},i,j)$ occurs before s' since the last preceding $done_i$, if one occurs. Therefore, by Lemma 11, a $rcv(\text{promise}(t),j,i)$ action must occur in α since the last $send(\text{bye}(m,t'),i,j)$ event (if one occurs). So eventually, $permission\text{-}from_i[j] < (\infty, n)$ for all $j \in \mathcal{I}$. We note that no action can occur at p_i in region T or W to cause an entry in the $permission\text{-}from_i$ array become (∞, n). Therefore, we have reached a contradiction. ■

Informally, the following lemma states that given certain simple conditions, processes cannot get stuck in their trying regions.

Lemma 13: Let α be a well-behaved execution of P, let s be some state in that execution, and fix $\mathcal{J} \subseteq \mathcal{I}$ such that $j \in \mathcal{J}$ iff $s.region_j \in \{\text{T,W}\}$ and for all states s' after s, $try\text{-}time_j$ does not increase beyond some time t. Then eventually an $mtext_j$ occurs after s for all $j \in \mathcal{J}$.

Proof: Assume (for contradiction) that there exists some set $\mathcal{J}' \subseteq \mathcal{J}$ such that no $mtext_j$ occurs after s' for all $j \in \mathcal{J}'$. If no $mtext_j$ action occurs, then it must be the case that no $ready_j$ action occurs: by definition of scheds(U_j), region-sequence(i,$\alpha|U_i$) may not end in R, so by well-formedness for j, a $mtext_j$ must follow every $ready_j$. Therefore, it suffices to show that a $ready_j$ occurs for some $j \in \mathcal{J}'$.

Let α' be the portion of α after which (1) all processes k not in $\mathcal{J}'$ have either increased $try\text{-}time_k$ beyond t or have issued $mtext_k$ actions, (2) all the processes $j \in \mathcal{J}$ have reached their maximum $try\text{-}time_j$[6], and (3) all the corresponding $rcv(\text{bye}(m,t''),k,l)$ and $rcv(\text{adv-promise}(t'),k,l)$ actions have occurred. Note that since *try-times* are tagged with process-ids, they are all unique. Now consider, among the $j \in \mathcal{J}'$, the one such that $try\text{-}time_j$ is least. Since all other processes k have either increased their *try-times* beyond $try\text{-}time_j$ or issued $mtext_k$ actions, and since all the corresponding $rcv(\text{adv-promise}(t'),k,l)$ and $rcv(\text{bye}(m,t''),k,l)$ actions have occurred, we know that for all $k \in \mathcal{I}$, $promises\text{-}to_j[k] \geq try\text{-}time_j$. Therefore, only $pending_j \neq \emptyset$ could prevent $ready_j$ from occurring.

However, since $try\text{-}time_i \geq clock_i$ in all states and for all k, $promises\text{-}to_j[k] \geq try\text{-}time_j$, nothing prevents $relay_j$ actions from occurring to empty $pending_j$. Therefore, a $ready_j$ must eventually occur, giving us a contradition. ■

The next two theorems correspond to Conditions (3a) and (3b) of schedule module M.

Theorem 14: Let α be a well-behaved execution of P. If a try_i occurs in α and each $relay_i$ thereafter is followed immediately by a try_i in $\alpha|U_i$, then a $ready_j$ occurs later in α.

Proof: Suppose (for contradiction) that no $ready_i$ occurs later in α. Since a try_i immediately follows each $relay_i$ in $\alpha|U_i$, no $mtext_i(\epsilon)$ can occur, by well-formedness for i. Therefore, by Lemma 12, there must exist a state s' in α such that $\forall j \in s.requested$, $s.permission\text{-}from_i[j] < (\infty, n)$, for all states s between s' and the next $ready_i$ action. Since we have assumed that no $ready_i$ action occurs, the property holds for all states after s'. Given this fact and the preconditions

[6] We know this must happen eventually because none can grow past t and *try-time* cannot be decreased in regions T or W.

on $ready_i$, there are only two ways in which the $ready_i$ action could not be enabled. Either $pending_i$ is not empty or $promises\text{-}to_i[j] < try\text{-}time_i$ for some j. If for all j, $promises\text{-}to_i[j] \geq try\text{-}time_i$, then nothing would prevent $relay_i$ actions from occurring to empty $pending_i$, since $try\text{-}time_i \geq clock_i$ by definition. Therefore, the only possibility is that forever after some point in α, $promises\text{-}to_i[j] < try\text{-}time_i$ for some (one or more) j. Since p_i only issues promises for times greater than $try\text{-}time_i$, there is a fixed number of processes j "in the way" of a $ready_i$ action. Furthermore, we know that none of these processes j eventually have $try\text{-}time_j$ larger than $try\text{-}time_i$, or else a $send$(adv-promise($try\text{-}time_j$),j,i) action would become enabled, and the corresponding rcv would eventually occur, causing $promises\text{-}to_i[j] > try\text{-}time_i$. Similarly, none of these processes may issue $mtext_j(m)$ actions, since that would enable a $send$(bye(m,t),j,i) action and the corresponding rcv would cause $promises\text{-}to_i[j] = (\infty, n)$. However, by Lemma 13, a $mtext_j$ must occur for each process j standing in the way, giving us our contradiction. ■

Theorem 15: Let α be an execution of P, where $\alpha|U_i \in \text{scheds}(U_i)$ for all $i \in \mathcal{I}$ and $\alpha|N \in \text{scheds}(N)$. If a $mtext_i(m \neq \epsilon)$ occurs in α and $try_i(S)$ is the last preceding try_i action in α, then a $relay_j(m)$ occurs later in α for each $j \in S$.

Proof: After $mtext_i(m \neq \epsilon)$ occurs in α, we know that a $ready_i$ must precede it, by well-formedness for i. So, by the preconditions of $ready_i$, all $j \in requested_i$, $permission\text{-}from_i < (\infty, n)$. Therefore, the actions $send$(bye(m,t),i,j) remain enabled until they occur. And by definition of N, the corresponding rcv(bye(m,t),i,j) actions must eventually occur.

Once rcv(bye(m,t),i,j) occurs, the only way for $relay_j(m)$ to be prevented is for $promises\text{-}to_j[k]$ to be less than t, for some $k \in \mathcal{I}$. Note any new promises granted by p_j must be greater than t until $relay_j(m)$ occurs, since $t \leq \max(pending)$. Therefore, by Theorem 14 and the result of the preceding paragraph, all promises granted by p_j for times less than t must eventually be relinquished. At that point $promises\text{-}to_j[k] \geq t, \forall k \in \mathcal{I}$, so eventually $relay_j(m)$ occurs. ■

Theorem 16: Module P solves schedule module M. *Proof:* Follows immediately from Theorems 3, 9, 10, 14, and 15 and the definition of M. ■

6 Complexity Analysis

In this section, we analyze the message and time complexities of the multicast protocol.

Let system A be the composition of modules $U_i, i \in \mathcal{I}$, module N, and modules $p_i, i \in \mathcal{I}$.

Let α be a execution of system A. We say that α is an *undeviating execution for* i iff every pair of actions $try_i(S)$ and $try(S')$ either have a $done_i$ between them or $S = S'$.

That is, in an undeviating execution for i, u_i does not "change its mind" about whether to issue a multicast message or which users to whom the the multicast should be sent.

6.1 Message Complexity:

There are four types of messages sent in the algorithm: req-promise, promise, adv-promise, and bye messages. If u_i issues $\pi = try_i(S)$ in an execution of system A, then we say that the following messages occur *as a result of* π: any requests by p_i for promises from any $p_j, j \in S$, any promises sent in response to those requests, any promise advancements by p_i to any $p_{i \in S}$, and any bye messages sent from p_i to $p_j, j \in S$. That is, we charge each try_i action with those messages required to complete the corresponding multicast.

Theorem 17: Let α be an undeviating execution for i, where $\alpha|U_i$ contains a $\pi = try_i(S)$. Then at most $4|S|$ network messages occur as a result of π.

Proof: By Lemma 4, we know that at most one *send*(req-promise,i,j), one *send*(promise(t),j,i) and one *send*(bye(m,t'),i,j) occur between π and the successful (or unsuccessful) completion of the multicast. Now we show that at most one *send*(adv-promise(t''),i,j) is required. Since the execution is undeviating, we do not require that any promises be requested (or received) from processes other than those named in S. Since no adv-promises are sent until promises are received from all in S, all promises need be advanced at most once, to the same logical time. ■

Note that in the nondeterministic algorithm we have presented, it is possible that more promises are requested than are actually needed. However, one could add a variable to keep track of a maximal set of processes that have been named in $try_i(S)$ actions since the last $done_i$ and require that promises only be requested from members of that set.

In executions that do not have the undeviating property, more messages may be required. In the worst case, the *try-set* grows by one with each try_i action until $|S| = n$, all promises granted by the new process exceed the old *try-time* and are received before the next try_i, and all promises are advanced after each promise is received. In this worst-case scenario, the number of req-promise, promise, and bye messages are the same as above, but the number of adv-promise messages is $O(n^2)$. However, one would expect such extreme behavior to be highly unlikely. (Alternative methods of promise advancement are outlined in Section 6.3.)

6.2 Time Complexity:

To analyze the time complexity, we need to make stronger assumptions than the eventuality conditions used for the liveness proofs. Let d be an upper bound on the time between a *send* event and the corresponding *rcv* (i.e., the message delay). We assume that process step time is insignificant in comparison to d. (For example, we assume that the time between receiving a request for a promise and sending the promise is negligable.)

First, we need to compute an upper bound on the time for a process with the lowest try-time to be able to send a multicast message once it has received all the necessary promises.

Lemma 18: Let α be an undeviating execution for i. If at real time r, p_i is in state s such that $s.promises\text{-}from_i[j] < (\infty, n)$ for all $j \in requested_i \cup s.tryset_i$ and p_i makes no requests for promises after state s. If p_i has the minimum *try-time* among all $j \in \mathcal{I}$ such that $s.region_j \in \{\text{T,W}\}$, then $ready_i$ occurs by time $r + 3d$.

Proof: Within time d after s, all *rcv*(bye(m,t),j,i) actions occur for all processes j such that $s.region_j \in \{\text{P,R,B}\}$ with $t < s.try\text{-}time_j$. Furthermore, for all processes j such that $s.region_j \in \{\text{T,W}\}$, actions *rcv*(adv-promise(t'),j,i), $t' > s.try\text{-}time_i$ occur within time $3d$ (one delay for p_j's promise requests, one delay for the promise messages, and one delay for the adv-promise message). Therefore, by time $r + 3d$, it is the case that $\min(promises\text{-}to_i) > s.try\text{-}time_i$. So, all the multicast messages waiting in $pending_i$ can be immediately relayed. Therefore, the preconditions for $ready_i$ are satisfied. ■

We say that p_i *depends on* p_j iff both have *region* $\in \{\text{T,R}\}$ and $try\text{-}time_i > promises\text{-}to_i[j]$. We say that p_i *indirectly depends on* p_j iff there is a sequence $p_i, p_1, p_2, \ldots, p_j$ such that p_i depends on p_1, p_1 depends on p_2, etc.

Lemma 19: Let α be an undeviating execution for all $i \in \mathcal{I}$. If at real time r, p_i is in state s such that $s.promises\text{-}from_i[j] < (\infty, n)$ for all $j \in requested_i \cup s.tryset_i$ and p_i makes no requests for promises after state s. Let z be the largest number of processes on which p_i indirectly depends between state s and the next $ready_i$. Then a $ready_i$ occurs by time $r + 4dz + 2d$

Proof: As before, by time $r + 3d$, $rcv(\text{adv-promise}(t'),j,i)$ actions occur for all p_j on which p_i depends. Furthermore, for any process p_k on which p_j depends (such that p_i depends on p_j), p_j receives p_k's promise request by time $r+2d$, for after that time p_j has received all its promises from the members of its *try-set*. Therefore, by time $r + d(z + 2)$, the *try-times* of all processes on which p_i indirectly depends have been determined. Consider p_l, the process with the lowest *try-time* on which p_j depends. By Lemma 18, we know that $ready_l$ occurs by time $r + d(z+2) + 3d$. Since any later promises received by p_l are sent after this $ready_i$, we know that p_i cannot again depend on p_l until after $ready_i$ occurs. We continue applying the same argument to the process p_m with the next lowest *try-time* until a $ready_i$ occurs. Therefore, $ready_i$ occurs by time $r + d(z + 2) + z(3d)$. ■

Thus, the time complexity depends on the concurrency inherent in the pattern of the multicast messages, since this is what determines the dependency order. Since z can be at most n, the delay is at most $4nd + 2d$.

Note that the worst-case time complexity matches one's expectations about what must happen when all n processes attempt to send multicast messages to every process. A simple inductive argument shows that any pessimistic protocol requires an $O(dn)$ delay in this worst-case scenario: Since all processes send to all other processes, the conditions of the problem require that the protocol enforce a total order on the multicasts. Thus, the process u whose message is the k^{th} message in the total order must wait at least $d(k - 1)$ time before sending its message, or else it could not have received all $k - 1$ messages ordered before it. (This, of course, assumes that all messages take the maximum time d to arrive.)

The worst-case scenario for an execution without the undeviating property is rather complicated. Process p_1, say, gives promises to all the other processes. Then, processes p_2 through p_n each change their minds n times about their *try-set* before finally performing multicasts in turn while p_1 waits. On receipt of p_n's multicast message, u_1 changes its mind about its *try-set* and issues a new try_i. But in the mean time, p_1 gives new promises to all the other processes $p_2, \ldots, p_n$. Then p_1 requests promises from its new *try-set* and, receiving those promises, advances its *try-time* past all the new promises it has granted. Thus, the same procedure can start over and repeat itself for a total of n times, since u_1 can change its mind at most n times before a $ready_i$ finally occurs. This worst-case scenario results in a delay of $O(n^3d)$. However, the scenario is, at the very least, unlikely.

For a deeper understanding of the true time complexity of the algorithm, one might state a measure of the concurrency inherent in the pattern of *try* actions and derive a time complexity in terms of that measure.

6.3 Possible Optimizations

To simplify the presentation of the algorithm, we chose to only send one message in a $relay_i$ action. And for the sake of generality, we chose to let p_i send itself messages over the network. As a minor modification, one might wish to send a sequence of messages. In addition, one might not want p_i to send any messages to itself.

A more significant modification would involve not waiting for promises requested from processes not in one's try set. That is, $ready_i$ would become enabled once promises have been received from all the processes named in the *try-set*, even if p_i has requested a promise that has not yet been received. Then p_i would send out "bye" messages to every process in *requested*, regardless of whether the promise had been received. This modification would require some mechanism for dealing with promises that come in late. One might keep track of the number of earlier $done_i$ actions and tag each request with that number; that tag would be appended to the corresponding promise by the granting process. In this way, promises arriving from an earlier multicast attempt could be ignored.

We mentioned earlier that there are other ways in which promise advancement might be handled. For example, one might not wish to wait until promises have been received from all the members in the *try-set* before advanding promises. Alternatively, one might a processes request promise advancement from those processes blocking its computation. More specifically, the following options are possible.

1. Advancement on demand: If a process p_j is in T with *try-time* $= t$, and has given a promise to p_i for a time t' less than t, then p_j may send p_i a message, asking it to advance the promise. Upon receiving such a message, if p_i has *try-time* $> t'$, then it will send p_j a promise advancement message.

2. Spontaneous advancement: This method allows p_i to nondeterministically send advancement messages when it notices that it is holding a promise with a time less than its *try-time*.

Deadlock avoidance methods similar to these are discussed in [Mi], although the problem studied there is different. In both cases, there is a trade-off between the message and time complexities: as one becomes more agressive about advancing promises to reduce time delays, the number of messages increases.

7 Impossibility Result

Recall that our correctness proof depended upon certain liveness assumptions about the user processes. Namely, the user processes are not allowed to stop at certain points in their executions. It turns out that that these assumptions are, in fact, necessary in order to solve the logically synchronous multicast problem.

In this section, we show that it is not possible to implement a solution to the logically synchronous multicast problem that tolerates even a single stopping fault of a user process. That is, we prove that there exists no wait-free implementation of a logically synchronous multicast protocol.

The proof proceeds by a reduction, using techniques developed by Herlihy [He]. We first demonstrate that logically synchronous multicast can be used to solve distributed consensus. We then appeal to the known result that distributed consensus cannot be solved in a wait-free manner [FLP].

The consensus problem is defined as follows. Consider n user processes $u_1,\ldots,u_n$, where each u_i has an initial value $v_i \in \{0,1\}$ and output actions *decide(0)* and *decide(1)* to announce its decision. A consensus protocol is correct iff it satisfies the following properties.

1. Agreement: If any user outputs *decide(v)*, then that is the only decision value of any process.

2. Validity: If all processes start with v, then v is the only possible decision value.

3. Termination: All processes eventually output some decision.

We say that an implementation is *wait-free* if a process can complete an operation within finite time, regardless of the execution speeds of the other processes. Equivalently, an implementation is *wait-free* if a process can eventually complete an operation even if some number of the other processes halt at arbitrary times.

For consensus, completing an operation means beginning the protocol and at some time later outputting a decision. For logically synchronous multicast, completing an operation means issuing a try_i output from region P and later receiving a $done_i$.

Lemma 20: If there exists a wait-free implementation of logically synchronous multicast, then there exists a wait-free implementation of distributed consensus.

Proof: Consider the following algorithm for reaching consensus among the n user processes $u_1,\ldots,u_n$ in system A, where each u_i has an initial value v_i in {0,1}. First, each process u_i issues $try_i(\mathcal{I})$. Upon receiving a $ready_i$, process u_i issues $mtext_i(v_i)$. Upon receiving its first $relay_i(v)$, process u_i uses v as its decision value. (After receiving a $relay_i$, if u_i is in region W, it issues a $mtext_i(\epsilon)$.)

Agreement: Since all multicast messages are delivered in the same relative order to all user processes, they all decide on the same value. Validity: If all users start with 0, then the decision will be 0, since that is the only value sent in a multicast by any user. Similarly if all users start with 1, then the decision will be 1. Termination: By the liveness condition, some process p_i will eventually issue a $ready_i$, and eventually all processes u_i will receive a $relay_i(v)$ for some v. ■

Theorem 21: There exists no wait-free implementation of logically synchronous multicast.

Proof: Suppose there were. Then by Lemma 20 there exists a wait-free implementation of distributed consensus. But it is known that there exists no wait-free implementation of distributed consensus [FLP]. ■

8 Conclusion

We have defined the logically synchronous multicast problem and presented a highly concurrent solution. To conclude the paper, we illustrate an application of this protocol in the area of distributed simulation. Namely, we consider distributed simulation of I/O automata.

The I/O automaton model has proven useful for describing algorithms and proving their correctness (for examples, see [Bl, FLS, GL, LG, LM, LMF, LMWF, LT1, LW, WLL]). Therefore, we believe that a simulation system based on that model would be a useful tool to aid in the study and understanding of complicated algorithms. *Distributing* the simulation, besides being an interesting exercise in itself, can also reduce the simulation time.

Recall from the definition of the I/O automaton model that input actions of automata are always enabled, and that an action shared by a set S of automata is the output of only one automaton and occurs simultaneously at all automata in S. In general, the actions enabled in a given state of an automaton may depend upon all previous actions occurring at that automaton. Furthermore, the fairness condition requires that given an automaton $\mathcal{A}$ and an execution α of $\mathcal{A}$, if some class $C \in \text{part}(\mathcal{A})$ has an action enabled in a state s of α, then either no action in C is enabled in some state s' occurring in α after s', or an action from C occurs in α after state s.

We wish to construct a distributed system for simulating fair executions of a given automaton $\mathcal{A}$, where $\mathcal{A}$ has some finite number of components $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$. To simplify the discussion, we shall assume that each component $\mathcal{A}_i$ has exactly one class in its partition. (The generalization allowing each component to have a finite number of classes is straightforward.)

We simply "plug in" a particular transition relation for each u_i in system A such that all of its schedules are in $scheds(U_i)$: We assign process u_i to simulate component $\mathcal{A}_i$. When $\mathcal{A}_i$ has an action π enabled, u_i may issue a $try_i(S)$ action, where S is the set of automata having π as an action.[7] Then, upon receiving a $ready_i$ input, u_i issues an $mtext(\pi)$, where π is the action associated with the previous try_i. Furthermore, u_i can issue a $mtext(\epsilon)$ only if no actions are enabled in $\mathcal{A}_i$. The $relay_i(\pi')$ input actions are used to drive the simulation of $\mathcal{A}_i$. When a $relay_i(\pi')$ action occurs, process u_i updates its state based on action π' occurring in $\mathcal{A}_i$.

[7] In a real implementation, one might have the system determine S based on π.

Given the schedule module M defined earlier, one can verify that the this distributed simulation satisfies the definitions of the model as described above. As far as each of the components of the simulation can tell, each action π occurring in the simulation happens simultaneously at every component having π in its signature. It is interesting to see how this construction and the liveness condition of the multicast problem work together to satisfy the fairness condition of the I/O automaton model.

Although the problem described in this paper has an application to the simulation system just described, we have presented it here as a general problem in a modular framework. The problem statement, the algorithm and its correctness proof, and the impossibility result are therefore general results, independent of any particular system or application.

Acknowledgements

I would like to thank Hagit Attiya and Jennifer Welch for suggesting the impossibility result, and Nancy Lynch and Mark Tuttle for their helpful comments on earlier drafts.

References

[Aw] Awerbuch, B. Complexity of Network Synchronization. JACM 32(4), October, 1985, pp. 804-823.

[Ba1] Bagrodia, R. A distributed algorithm to implement the generalized alternative command of CSP. *The 6th International Conference on Distributed Computing Systems*, May 1986, pp. 422-427.

[Ba2] Bagrodia, R. A distributed algorithm to implement N-party rendezvous. *The 7th Conference on Foundations of Software Technology and Computer Science*, Pune, India, December 1987. Lecture Notes in Computer Science 287, Springer Verlag, 1987.

[BJ] Birman, K.P., and Joseph, T.A. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[Bl] Bloom, B. Constructing Two-Writer Atomic Registers. *6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Vancouver, British Columbia, Canada, August, 1987, pp. 249-259. Also, to appear in Special Issue of IEEE Transactions on Computing, on Parallel and Distributed Algorithms.

[DoD] Department of Defense, Ada Programming Language, ANSI/MIL-STD-1815A-1983.

[FLS] Fekete, A., Lynch, N., and Shrira, L. A Modular Proof of Correctness for a Network Synchronizer. *2nd International Workshop on Distributed Algorithms,* Amsterdam, The Netherlands, July,1987.

[FLP] Fischer, M., Lynch, N., and Paterson, M. Impossibility of distributed consensus with one family faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[GL] Goldman, K.J., and Lynch, N.A. Quorum Consensus in Nested Transaction Systems. *6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,*Vancouver, British Columbia, Canada, August, 1987.

[He] Herlihy, M. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Toronto, Ontario, Canada, August, 1988, pp. 276-290.

[Ho] Hoare, C. A. R. Communicating Sequential Processes. Prentice-Hall, 1985.

[La] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, 1978.

[LG] Lynch, N.A., and Goldman, K.J. Distributed Algorithms. MIT Research Seminar Series MIT/LCS/RSS-5, May 1989.

[LM] Lynch, N.A., and Merritt, M. Introduction to the Theory of Nested Transactions. *ICDT'86 International Conference on Database Theory.* Rome, Italy, September, 1986, pp. 278-305. Also, MIT/LCS/TR-367 July 1986. A revised version will appear in *Theoretical Computer Science.*

[LMF] Lynch, N., Mansour, Y., and Fekete, A. Data Link Layer: Two Impossibility Results. In *Proceedings of 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Toronto, Ontario, Canada, August, 1988, pp. 149-170.

[LMWF] Lynch, N., Merritt, M., Weihl, W., and Fekete, A. Atomic Transactions. In progress.

[LT1] Lynch, N.A., and Tuttle, M.R. Hierarchical Correctness Proofs for Distributed Algorithms. Master's Thesis, Massachusetts Institute of Technology, April, 1987. MIT/LCS/TR-387, April, 1987.

[LT2] Lynch, N.A., and Tuttle, M.R. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Vancouver, British Columbia, Canada, August, 1987, pp. 137-151.

[LT3] Lynch, N.A., and Tuttle, M.R. An Introduction to Input/Output Automata. *CWI Quarterly*, CWI Amsterdam, September 1989, to appear.

[LW] Lynch, N.A., and Welch, J.L. Synthesis of Efficient Drinking Philosophers Algorithms. In progress.

[Mi] Misra, J. Distributed Discrete-Event Simulation. *Computing Surveys*, 18(1):39–65, 1986.

[WLL] Welch, J., Lamport, L., and Lynch, N. A Lattice-Structured Proof of a Minimum Spanning Tree Algorithm. In *Proceedings of 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Toronto, Ontario, Canada, August, 1988, pp. 28-43.