Distributed Computing (1991) 4:189–207

DISTRIBUTED COMPUTING

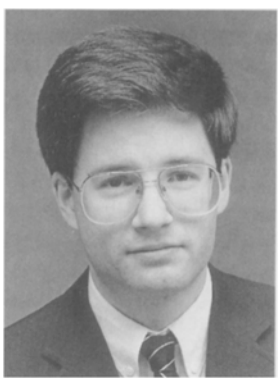


# Highly concurrent logically synchronous multicast*

**Kenneth J. Goldman**

Department of Computer Science, Washington University, St. Louis, MO 63130, USA

Received July 3, 1989 / Accepted November 11, 1990



**Kenneth J. Goldman** received the Sc.B. degree in Computer Science from Brown University in 1984, the S.M. degree in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology in 1987, and the Ph.D. degree in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology in 1990. As part of his doctoral work, he designed and implemented the Spectrum Simulation System, a distributed algorithm development tool based on the I/O automaton model of Lynch and Tuttle. His publications include papers in the areas of models for distributed computing, database concurrency control, human interfaces, and image processing. He is currently Assistant Professor in the Department of Computer Science at Washington University in St. Louis.

**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 careful problem specification stated in the I/O automaton model. We conclude the paper by describing how the logically synchronous multicast protocol can be used for distributed simulation of algorithms expressed as I/O automata.

**Key words:** Distributed algorithms – Multicast – Synchronization – Logical time – Input/output automata

## 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) There exists a total order on all multicasts in a computation such that the delivery order of multicast messages at each process is consistent with that total order.
(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.

The first two conditions say 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

* A preliminary version of this paper appeared in the Proceedings of the 3rd International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 392, Bermond and Raynal, Eds., Springer-Verlag, Berlin, 1989, pp 94–109.

This research was conducted at the Massachusetts Institute of Technology Laboratory for Computer Science and was supported in part by the National Science Foundation under Grant 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

would expect that the communication costs of an algorithm to solve this problem would be independent of $n$. We present a solution that takes advantage of the concurrency inherent in the problem and requires at most $4|S|$ messages per multicast, provided that a process does not "change its mind" about the set of participants.

Various approaches to ordering messages in asynchronous systems have been studied. Lamport [19] uses logical clocks to produce a total ordering on messages. Birman and Joseph [5] present several types of fault-tolerant protocols, where failures are assumed to be detectable by timeouts. 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 procees broadcasts 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. Broadcast protocols may be used to achieve process synchronization in distributed systems. For example, Schneider presents a synchronization technique that assumes a process may reliably broadcast a message to all other running processes such that messages originating at a given process are received by other processes in the order sent [28]. Joseph and Birman provide an extensive dicussion of reliable broadcast protocols in [18].

Like ours, the protocols of both [19] and [5] assign a global ordering to messages. 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 at 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).

Motivated by CSP [17] and ADA [1], multiway handshake protocols have been studied extensively. (For examples, see [3], [4], and [7]). These protocols must 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 any participant in a handshake to block the handshake from occurring. Since a decision about whether to accept or refuse a handshake may depend (in general) on all earlier events, each process can be involved in scheduling at most one handshake at a time. For example, let event $e$ be a handshake having participant processes $p_1$ and $p_2$. Process $p_1$ cannot permit process $p_2$ to complete event $e$ until $p_1$ knows that no event $e'$ to be ordered before $e$ will cause $e$ to be refused by $p_1$. In general, $p_1$ cannot permit $p_2$ to complete $e$ until all events at $p_1$ ordered before $e$ have already occurred. Our problem admits more concurrency, since a process cannot refuse to accept a multicast message. Whether or not a multicast occurs is entirely under the control of the sender. Therefore, a process can permit many multicasts destined for it to proceed concurrently[1].

One interesting feature of our problem is that it lies between the two general approaches described above. As we have described, it permits concurrent scheduling of events, yet imposes a strong, useful structure on the message delivery order.

Other related work includes papers by Awerbuch [2] and Misra [26], which study different problems in the area of simulating synchronous systems on asynchronous ones. In both cases, however, the computational models being simulated are very different from ours. Awerbuch's goal is to take algorithms written for systems in which processes proceed in lock step, and simulate them on systems in which processes proceed asynchronously. An algorithm is presented for generating "pulse" messages to synchronize the computation. In contrast, the purpose of logically synchronous multicast is to provide the illusion of synchronous communication among dynamically changing subsets of processes, as opposed to synchronized steps at all processors. Misra [26] studies the problem of distributed discrete event simulation. One important difference between Misra's work and logically synchronous multicast is that Misra fixes the communication pattern. This gives the problem additional structure, since each process expects messages only from a (small) fixed subset of the other processes. In the present work, we assume that a process may potentially receive a multicast from any other process in the system. In spite of this difference, some of Misra's techniques, particularly those for breaking deadlock, can be applied to our problem. This is discussed in Sect. 6.3.

The remainder of the paper is organized as follows. Section 2 provides a brief introduction to the I/O automaten model. In Sect. 3, we present the architecture of the logically synchronous multicast problem and a statement of correctness in terms of the model. In Sect. 4, we formally present the algorithm using the I/O automaton model. In Sect. 5 and 6, we give a complete correctness proof and analyze the message and time complexities.

The author has recently developed a simulation system for algorithms expressed as systems of I/O automata [14]. The logically synchronous multicast problem was motivated by a desire to distribute the simulation 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 [24, 25]. We have chosen this model because it encourages precise statements of the problems to be solved by modules in concurrent

[1] These comments apply only to *pessimistic* protocols, in which no rollback is allowed. If rollback is permitted, an *optimistic* strategy for CSP-style synchronization could be achieved with more concurrency, but at the expense of the overhead necessary for rollback

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 [25], 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 modelling 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 $A$ 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 $\pi$ there is a transition $(s', \pi, s)$ 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. Each class of the parition may be thought of as a separate process. We refer to an element $(s', \pi, s)$ of $steps(A)$ as a *step* of $A$. If $(s', \pi, s)$ is a step of $A$, then $\pi$ 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 $s_0, \pi_1, s_1, \ldots, \pi_n, s_n$ 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 $\alpha$ is the subsequence of $\alpha$ consisting of the actions appearing in $\alpha$. The *behavior* of an execution or schedule $\alpha$ of $A$ is the subsequence of $\alpha$ consisting of *external* actions. The sets of executions, finite executions, schedules, finite schedules, behaviors, and finite behaviors are denoted $execs(A)$, $finexecs(A)$, $scheds(A)$, $finscheds(A)$, $behs(A)$, and $finbehs(A)$, respectively. 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 modelling a complex system by composing automata modelling the simpler system components. When we compose a collection of automata, we identify an output action $\pi$ of one automaton with the input action $\pi$ of each automaton having $\pi$ as an input action. Consequently, when one automaton having $\pi$ as an output action performs $\pi$, all automata having $\pi$ as an action perform $\pi$ simultaneously (automata not having $\pi$ as an action do nothing).

Since we require that at most one system component controls the performance of any given action, we must place some compatibility restrictions on the collections of automata that may be composed. A countable collection $\{S_i\}_{i\in I}$ of action signatures is said to be *strongly compatible* if for all $i, j\in I$ satisfying $i\neq j$ we have

1. $out(S_i)\cap out(S_j)=\emptyset$,
2. $int(S_i)\cap acts(S_j)=\emptyset$, and

no action is contained in infinitely many sets $acts(S_i)$, $i\in I$. We say that a collection of automata are *strongly compatible* if their action signatures are strongly compatible.

The *composition* $S=\prod_{i\in I} S_i$ of a countable collection of strongly compatible action signatures $\{S_i\}_{i\in I}$ is defined to be the action signature with

- $in(S)=\bigcup_{i\in I} in(S_i)-\bigcup_{i\in I} out(S_i)$,
- $out(S)=\bigcup_{i\in I} out(S_i)$, and
- $int(S)=\bigcup_{i\in I} int(S_i)$.

The *composition* $A=\prod_{i\in I} A_i$ of a countable collection of strongly compatible automata $\{A_i\}_{i\in I}$ is the automaton defined as follows[2].

- $sig(A)=\prod_{i\in I} sig(A_i)$,
- $states(A)=\prod_{i\in I} states(A_i)$,
- $start(A)=\prod_{i\in I} start(A_i)$,
- $steps(A)$ is the set of triples $(\vec{s}_1, \pi, \vec{s}_2)$ such that, for all $i\in I$, if $\pi\in acts(A_i)$ then $(\vec{s}_1[i], \pi, \vec{s}_2[i])\in steps(A_i)$, and if $\pi\notin acts(A_i)$ then $\vec{s}_1[i]=\vec{s}_2[i]$, and
- $part(A)=\bigcup_{i\in I} part(A_i)$.

Given an execution $\alpha=\vec{s}_0\pi_1\vec{s}_1\ldots$ of $A$, let $\alpha|A_i$ (read "$\alpha$ projected on $A_i$") be the sequence obtained by deleting $\pi_j\vec{s}_j$ when $\pi_j\notin acts(A_i)$ and replacing the remaining $\vec{s}_j$ by $\vec{s}_j[i]$.

[2] Here $start(A)$ and $states(A)$ are defined in terms of the ordinary Cartesian product, while $sig(A)$ is defined in terms of the composition of actions signatures just defined. Also, we use the notation $\vec{s}[i]$ to denote the $i$th component of the state vector $\vec{s}$

### 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 or processes) 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 $\alpha$ of $A$ such that the following conditions hold for each class $C$ of $part(A)$:

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

We denote the set of fair executions of $A$ by $fairexecs(A)$. We say that $\beta$ is a *fair behavior* of $A$ if $\beta$ is the behavior of a fair execution of $A$, and we denote the set of fair behaviors of $A$ by $fairbehs(A)$. Similarly, $\beta$ is a *fair schedule* of $A$ if $\beta$ is the schedule of a fair execution of $A$, and we denote the set of fair schedules of $A$ by $fairscheds(A)$.

### 2.4 Problem specification

A 'problem' to be solved by an I/O automation is formalized as a set of (finite and infinite) sequences of external actions. An automaton is said to *solve* a problem $P$ provided that it 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 set $scheds(H)$ is the set of sequences $\beta$ 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 set of sequences $\mathscr{P}$ is said to be *prefix-closed* if $\beta\in\mathscr{P}$ whenever both $\beta$ is a prefix of $\alpha$ and $\alpha\in\mathscr{P}$. A module $M$ (either an automaton or schedule module) is said to be *prefix-closed* provided that $finbehs(M)$ is prefix-closed. Let $M$ be a prefix-closed module and let $\mathscr{P}$ be a nonempty, prefix-closed set of sequences of actions from a set $\Phi$ satisfying $\Phi\cap int(M)=\emptyset$. We say that M *preserves* $\mathscr{P}$ if $\beta\pi|\Phi\in\mathscr{P}$ whenever $\beta|\Phi\in\mathscr{P}$, $\pi\in out(M)$, and $\beta\pi|M\in finbehs(M)$. Informally, a module *preserves* a property $\mathscr{P}$ iff the module is not the first to violate $\mathscr{P}$: as long as the environment only provides inputs such that the cumulative behavior satisfies $\mathscr{P}$, the module will only perform outputs such that the cumulative behavior satisfies $\mathscr{P}$. 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 logically synchronous multicast problem and then present a schedule module to define correctness for a multicast protocol.

### 3.1 The architecture

Let $\mathscr{I}=\{1, \ldots, n\}$. Let $\mathscr{S}$ denote a universal set of text strings (containing the empty string $\varepsilon$), and let $\mathscr{M}$ denote a universal set of messages. Let $u_i$, $i\in\mathscr{I}$, denote the $n$ user processes engaged in the computation, and let $p_i$, $i\in\mathscr{I}$, denote $n$ additional processes. Together, the $p_i$'s are to solve the multicast problem, where 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 boundaries between $u_i$ and $p_i$ and between $p_i$ and the network are defined by several actions, as illustrated in Fig. 1. 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 has occurred most recently. (We will formalize

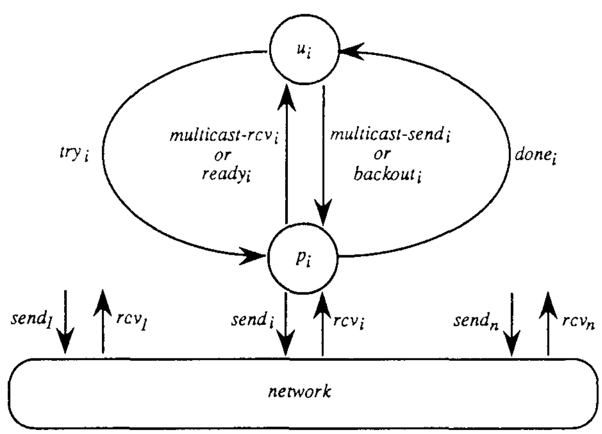



**Fig. 1.** System architecture. Arguments of actions are omitted

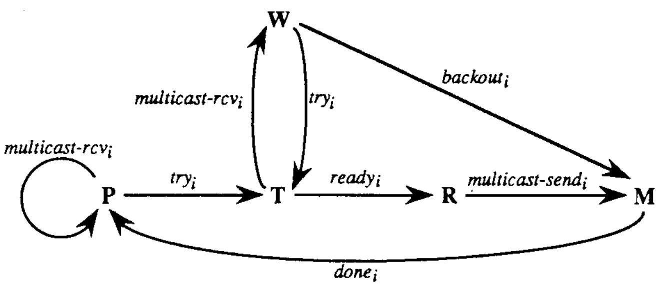



**Fig. 2.** Region changes for $p_i$

this later). Figure 2 illustrates the 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 $try_i(S \subseteq \mathcal{I})$[3] action, indicating that $u_i$ would like to send a multicast message to 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). The $ready_i$ action constitutes permission for $u_i$ to actually send the multicast. That is, after receiving the $ready_i$ action as input, user $u_i$ may issue a $multicast\text{-}send_i\,(m \in \mathcal{S})$ action, where the argument indicates the desired text of the multicast message. Upon receiving the $multicast\text{-}send_i$ action, $p_i$ is said to enter its "multicast" region (M), where it completes the multicast and returns to region P by issuing a $done_i$ action. Region M is present to ensure that each multicast for $u_i$ is completed before the next multicast is requested by $u_i$.

In addition to these actions, there are $multicast\text{-}rcv_i\ (m \in \mathcal{S})$ 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 $multicast\text{-}rcv_i$ action while in T[4]. In W, $p_i$ waits to see if $u_i$ has "changed its mind" about its own multicast after hearing the information contained in the $multicast\text{-}rcv_i$ 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 a $backout_i$ action. A $backout_i$ action sends $p_i$ to region $M$ (rather than directly to region $P$) so that $p_i$ may "clean up" from the failed multicast attempt before the next $try_i$ action occurs.

It might seem that one could eliminate region W and the $backout_i$ actions by having $multicast\text{-}rcv_i$ actions take $p_i$ to region P. However, this would make it difficult to express the liveness notion that $u_i$ eventually must 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". As a separate modification of this architecture, one might consider elimination of the $ready_i$ and $multicast\text{-}send_i$ actions in favor of including the desired text of the multicast as a second argument to the $try_i$ actions. However, as we will see, the $ready_i$ and $multicast\text{-}send_i$ actions serve as useful "commit" points in stating both the safety and liveness conditions of the problem. They also provide a convenient way to separate the successful multicasts from the unsuccessful $try_i$ attempts in reasoning about algorithm executions.

### 3.2 Correctness

Since the only actions under the control of the protocol are the outputs of the $p_i$'s, we only wish to require that the protocol behaves correctly when its environment, namely the composition of the $u_i$'s and the network, is well-behaved. To this end, we define schedule modules that specify the allowable behaviors of each $u_i$ and the network. Based on these, 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:

$$\mathrm{in}(U_i) = \{multicast\text{-}rcv_i(m \in \mathcal{S}), ready_i, done_i\}$$
$$\mathrm{out}(U_i) = \{try_i(S \subseteq \mathcal{I}), multicast\text{-}send_i(m \in \mathcal{S}), backout_i\}$$

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 that makes use of this definition. Let the alphabet $\Sigma = \{$P, T, R, M, W, X$\}$. Let $\alpha$ be an arbitrary sequence of actions. We define the *region of i after* $\alpha$, denoted $r(i, \alpha)$, to be an element of $\Sigma$ defined recursively as follows. If $\alpha | U_i$ is empty ($\varepsilon$), then $r(i, \alpha) = \mathrm{P}$. If $\alpha = \alpha'\pi$, then, ignoring arguments to action names,

$$r(i,\alpha) = \begin{cases} r(i,\alpha') & \text{if } \pi \notin acts(U_i), \\ \mathrm{P} & \text{if } (\pi = done_i \wedge r(i,\alpha') = \mathrm{M}) \\ & \quad \vee (\pi = multicast\text{-}rcv_i \wedge r(i,\alpha') = \mathrm{P}), \\ \mathrm{T} & \text{if } \pi = try_i \wedge r(i,\alpha') \in \{\mathrm{P}, \mathrm{W}\}, \\ \mathrm{R} & \text{if } \pi = ready_i \wedge r(i,\alpha') = \mathrm{T}, \\ \mathrm{M} & \text{if } (\pi = multicast\text{-}send_i \wedge r(i,\alpha') = \mathrm{R}) \\ & \quad \vee (\pi = backout_i \wedge r(i,\alpha') = \mathrm{W}), \\ \mathrm{W} & \text{if } \pi = multicast\text{-}rcv_i \wedge r(i,\alpha') = \mathrm{T}, \\ \mathrm{X} & \text{otherwise.} \end{cases}$$

Given an arbitrary action sequence $\alpha$ and an index $i \in \mathcal{I}$, we define the *region sequence for i in* $\alpha$, denoted *region-sequence*$(i, \alpha)$, to be the concatenation of $r(i, \alpha')$ for each prefix of $\alpha$ in order, starting with $r(i, \varepsilon)$ and ending with $r(i, \alpha)$. Note close correspondence between Fig. 2 and the definition of region-sequence.

Let $\alpha$ be an arbitrary sequence of actions. We say that $\alpha$ is *user well-formed for i* iff

1. for all $try_i(S)$ actions in $\alpha$, $i \in S$, and
2. *region-sequence*$(i, \alpha)$ does not contain the symbol X.

We can define the set of schedules for $U_i$. Let $\alpha$ be a sequence of actions in sig $(U_i)$. Then $\alpha \in scheds(U_i)$ iff

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

[4] A $multicast\text{-}rcv_i$ action from region P does not cause a region change

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

The first property is used to help define the safety conditions for the logically synchronous multicast problem, since a multicast protocol must perform correctly only if its environment is well behaved. The second property, used in defining the liveness conditions, says that a user process cannot "stop" in regions W or R. This is used to express the notion that a multicast protocol must guarantee progress only if users trying to send multicasts eventually respond to *multicast-rcv* and *ready* actions.

We define schedule module $U$ to be the composition $\prod_{i \in \mathscr{I}} U_i$.

**Schedule module *N*.** We now define a schedule module specifying the network. The signature is as follows:

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

To define the allowable schedules of the network, we use a *correspondence relation* similar to that of [10]. A correspondence relation between the *send* and *rcv* events in a sequence captures the correspondence between the send and receipt of a message. Consider the following properties that may hold for a particular correspondence relation for a given sequence $\alpha$:

(S1) $\forall i_1, i_2, j_1, j_2 \in \mathscr{I}$, $\forall m_1, m_2 \in \mathscr{M}$, if event $\pi_1 = send(m_1, i_1, j_1)$ corresponds to event $\pi_2 = rcv(m_2, i_2, j_2)$, then $m_1 = m_2$, $i_1 = i_2$, $j_1 = j_2$, and $\pi_1$ precedes $\pi_2$ in $\alpha$.
(S2) $\forall i, j \in \mathscr{I}$, $\forall m \in \mathscr{M}$, each $rcv(m, i, j)$ corresponds to exactly one $send(m, i, j)$.
(S3) $\forall i, j \in \mathscr{I}$, $\forall m \in \mathscr{M}$, each $send(m, i, j)$ corresponds to at most one $rcv(m, i, j)$.
(S4) $\forall i, j \in \mathscr{I}$, $\forall m, m' \in \mathscr{M}$, if event $rcv(m, i, j)$ occurs in $\alpha$ before event $rcv(m', i, j)$, then their corresponding events $send(m, i, j)$ and $send(m', i, j)$ occur in the same order.
(L) $\forall i, j \in \mathscr{I}$, $\forall m \in \mathscr{M}$, each $send(m, i, j)$ event has a corresponding $rcv(m, i, j)$ event.

The first four properties (S1–S4) are *safety* properties. They say that a message is delivered only after it is sent, that no spurious messages are delivered, that a message is delivered at most once (for each time it is sent), and that messages between a pair of processes are delivered in the order sent. Property (L) is a *liveness* property; it says that each message sent is eventually delivered.

If $\alpha$ is a sequence of actions of $N$, we say that $\alpha$ is *network well-formed* iff there exists a correspondence relation for $\alpha$ that satisfies properties S1–S4. Moreover, $\alpha \in scheds(N)$ iff the correspondence relation also satisfies property (L). Property (L) will be used only in the liveness proof.

**Schedule module M.** The correctness conditions for the logically synchronous multicast problem can now be stated formally in terms of the actions at the boundaries of the user processes. We do this with a schedule module $M$ that defines the multicast problem. We define the signature of $M$ as follows:

$$\mathrm{in}(M) \;=\mathrm{out}(U) \cup \mathrm{out}(N)$$
$$\mathrm{out}(M)=\mathrm{in}(U) \cup \mathrm{in}(N)$$

In defining the schedules of $M$, we use a correspondence relation technique (similar to the one used to define schedule module $N$) to capture the correspondence between each *multicast-send* event and the resulting *multicast-rcv* events. Let $\alpha$ be a sequence of actions of sig$(M)$, and let correspondence relation $\mathscr{C}$ relate the *multicast-send* and *multicast-rcv* events of $\alpha$. We say that $\mathscr{C}$ is a *proper correspondence relation for* $\alpha$ iff it satisfies the following properties:

1. $\forall i, j \in \mathscr{I}$, $\forall m, m' \in \mathscr{S}$, if event $\pi_1 = multicast\text{-}send_i(m)$ corresponds to event $\pi_2 = multicast\text{-}rcv_j(m')$, and $try_i(S)$ is the last $try_i$ action in $\alpha$ before $\pi_1$, then $m = m'$ and $j \in S$.
2. $\forall i, j \in \mathscr{I}$, $\forall m \in \mathscr{S}$, each $multicast\text{-}rcv_j(m)$ corresponds to exactly one $multicast\text{-}send_i(m)$.
3. $\forall i, j \in \mathscr{I}$, $\forall m \in \mathscr{S}$, each $multicast\text{-}send_i(m)$ corresponds to at most one $multicast\text{-}rcv_j(m)$.

Informally, these properties say that (1) a $multicast\text{-}rcv_j(m)$ must contain the same text argument as its corresponding send, and that $j$ must name one of the destination processes, (2) a *multicast-rcv* event corresponds to exactly one *multicast-send*, and (3) a given *multicast-send* event corresponds to at most one $multicast\text{-}rcv_j$ for each possible destination process $u_j$.

Let $\alpha$ be a sequence of actions of sig$(M)$, let $\mathscr{C}$ be a proper correspondence relation for $\alpha$, and let $\prec$ be a total order on all *multicast-send* events in $\alpha$. We say that $\prec$ is a *proper total order for* $\mathscr{C}$ *and* $\alpha$ *iff* the following property holds: $\forall i, j, k \in \mathscr{I}$, $m, m' \in \mathscr{S}$, if $multicast\text{-}send_i(m)$ and $multicast\text{-}send_j(m')$ occur in $\alpha$ with corresponding receives $multicast\text{-}rcv_k(m)$ and $multicast\text{-}rcv_k(m')$ and if $\prec$ orders $multicast\text{-}send_i(m)$ before $multicast\text{-}send_j(m')$, then $multicast\text{-}rcv_k(m)$ occurs in $\alpha$ before $multicast\text{-}rcv_k(m')$. Informally, this says the order of multicast deliveries at each user process must be consistent with the total order $\prec$. One may notice that a proper total order is not necessarily consistent with the order of multicasts sent by each individual process. This consistency requirement is handled separately by condition (2c) below.

Let $\alpha$ be a sequence of actions of sig$(M)$. Then $\alpha \in scheds(M)$ iff there exists a correspondence relation $\mathscr{C}$ and total order $\prec$ such that the following conditions hold.

1. $\forall i \in \mathscr{I}$, $M$ preserves user well-formedness for $i$ in $\alpha$.
2. If $\alpha$ is user well-formed for every $i \in \mathscr{I}$ and $\alpha$ is network well-formed, then
   (a) $\mathscr{C}$ is a proper correspondence relation for $\alpha$,
   (b) $\prec$ is a proper total for $\mathscr{C}$ and $\alpha$, and
   (c) $\forall m \in \mathscr{S}$, if $\pi = multicast\text{-}send_i(m)$ occurs in $\alpha$, then no $multicast\text{-}rcv_i$ occurs between $\pi$ and the $multicast\text{-}rcv_i(m)$ corresponding to $\pi$.
3. If $\alpha | N \in scheds(N)$ and $\forall i \in \mathscr{I}$, $\alpha | U_i \in scheds(U_i)$, then the following hold:
   (a) $\forall i \in \mathscr{I}$, if a $try_i$ occurs in $\alpha$, then either a $backout_i$ or a $ready_i$ occurs later in $\alpha$.

(b) $\forall i \in \mathscr{I}$, $\forall S \subseteq \mathscr{I}$, if a *multicast-send*$_i(m)$ occurs in $\alpha$ and $try_i(S)$ is the last preceding $try_i$ action in $\alpha$, then a corresponding *multicast-rcv*$_j(m)$ occurs later in $\alpha$ for each $j \in S$.

Items (1) and (2) are the required *safety* properties. Part (2c) is needed to ensure that user processes have "up to date" information when sending a multicast message. This also ensures that multicast messages sent by a given process are delivered in the order sent. Item (3) is the required *liveness* property. Part (3a) says that if a user process does not back out of its attempt to perform a multicast, then eventually it will receive permission to send the multicast. Part (3b) says that if a multicast is sent by a user process, then eventually all destination user processes will receive it. Note that the hypothesis of item (3) is needed to ensure that liveness properties hold for the users and the network. That is, we require that a solution to the multicast problem guarantee progress only if the users and the network satisfy their liveness requirements, namely that every user responds to *multicast-rcv* and *ready* actions and that every message is eventually delivered. 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 $p_i$, $i \in \mathscr{I}$. We show in Sect. 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 $\mathscr{T} = (\{1, 2, \ldots\} \cup \infty) \times \mathscr{I}$, 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[5]. The process $p_i$ delivers all multicast messages destined for $u_i$ in logical time order.

The state of each automaton $p_i$ has several components. The variable *region*$\in\{$P, T, W, R, B$\}$ is initially set to P and holds the current region of $p_i$ as described in Sect. 3.1. The variables *try-set, need-set, requested,* and *requests* are subsets of $\mathscr{I}$, initially empty. The *try-set* names the processes to whom $u_i$ would like to send a multicast, and the *need-set* contains the union of all values of *try-set* since $p_i$ was last in region P. The two sets *requested* and *requests* name the processes to whom $p_i$ has sent requests for "promises" and the processes from whom $p_i$ has received such requests. We will explain promises shortly. The variable *text*$\in\mathscr{S}$ is initially undefined, and is used to hold the text of the latest multicast by $u_i$. Two arrays of logical times indexed by $\mathscr{I}$ are kept: *promises-to* and *promises-from.* The entries of these arrays, initially $(\infty, n)$, are used to keep track of the times of promises granted and received, respectively. Two additional logical time variables, *clock* and *mctime*, are initially $(0, i)$. The *clock* contains the time of latest multicast received by $u_i$, and *mctime* contains the time of the latest multicast sent by $u_i$. Finally, the variable *pending* is an initially empty set of (text$\in\mathscr{S}$, time$\in\mathscr{T}$) pairs. This set contains all multicast messages received by $p_i$ but not yet delivered to $u_i$.

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

The transition relation for $p_i$ is shown in Fig. 3. "P" and "E" denote precondition and effect, respectively. An action is enabled in exactly those states $s'$ for which the precondition is satisfied. If an action has no precondition, it is enabled in all states. When an action occurs, $p_i$'s new state $s$ is determined according to the statements in the effects clause. States $s$ and $s'$ agree on components not assigned values in the effects clause. Automaton $p_i$ has the following signature.

| | |
|---|---|
| Input actions: | $try_i(S \subseteq \mathscr{I})$ |
| | $backout_i$ |
| | *multicast-send*$_i(m \in \mathscr{S})$ |
| | $rcv(m \in \mathscr{M}, j \in \mathscr{I}, i)$ |
| Output actions: | *multicast-rcv*$_i(m \in \mathscr{S})$ |
| | $ready_i$ |
| | $done_i$ |
| | $send(m \in \mathscr{M}, i, j \in \mathscr{I})$ |

The equivalence classes of $part(p_i)$ are as follows. The actions *multicast-rcv*$_i$, $ready_i$, and $done_i$ are together in one class. And for each $j \in \mathscr{I}$, there exist four classes containing the sets of actions $send(\text{promise}(t \in \mathscr{T}), i, j)$, $send(\text{req-promise}, i, j)$, $send(\text{adv-promise}(t \in \mathscr{T}), i, j)$, and $send(\text{multicast}(m \in \mathscr{S}, t \in \mathscr{T}), i, j)$. This choice of a partition simplifies reasoning about what actions must eventually occur in an execution. However, the necessary liveness properties could also be guaranteed with only two classes: one for $send(\text{promise}(t \in \mathscr{T}), i, j)$ actions, using a queue to ensure fairness to each $j$, and one class for all remaining output actions.

To describe the logically synchronous multicast protocol, we chronicle the events that take place between $u_i$'s multicast request and the completion of the multicast. To more fully understand this description, it is recommended that the reader follow along in the code for $p_i$ given in Fig. 3. Unless otherwise noted, the word "process" refers to one of the processes $p_i$, $i \in \mathscr{I}$. Also, we use the words "time" and "logical time" interchangeably.

To initiate the request to perform a multicast, $u_i$ issues a $try_i(S)$ action, where $S$ is the set of indices of user processes that are to receive the multicast.[6] The

[5] We never use $\infty$ in the time of a multicast message; it is used only as a place holder

[6] Recall from the definition of $U_i$ that $i \in S$

Input Actions:

– $try_i(S)$
  E: $s.try\text{-}set = S$
  $s.need\text{-}set = s'.need\text{-}set \cup S$
  $s.region = \mathrm{T}$

– $rcv(\text{req-promise}, j \in \mathscr{I}, i)$
  E: $s.requests = s'.requests \cup \{j\}$

– $rcv(\text{promise}(t \in \mathscr{T}), j \in \mathscr{I}, i)$
  E: $s.promises\text{-}from[j] = t$

– $multicast\text{-}send_i(m)$
  E: $s.text = m$
  $s.region = \mathrm{M}$

– $rcv(\text{multicast}(m \in \mathscr{S}, t \in \mathscr{T}), j \in \mathscr{I}, i)$
  E: $s.promises\text{-}to[j] = (\infty, n)$
  **if** $m \neq \varepsilon$ **then**
  $s.pending = s'.pending \cup \{(m, t)\}$

– $backout_i$
  E: $s.try\text{-}set = \emptyset$
  $s.region = \mathrm{M}$

– $rcv(\text{adv-promise}(t \in \mathscr{T}), j \in \mathscr{I}, i)$
  E: $s.promises\text{-}to[j] = t$

Output Actions:

– $send(\text{req-promise}, i, j \in \mathscr{I})$
  P: $s'.region \in \{\mathrm{T}, \mathrm{W}\}$
  $j \in s'.need\text{-}set \setminus s'.requested$
  E: $s.requested = s'.requested \cup \{j\}$

– $send(\text{promise}(t \in \mathscr{T}), i, j \in \mathscr{I})$
  P: $j \in s'.requests$
  $t > \max(lb\text{-}time(s'), \max(s'.pending).\mathrm{time})$
  E: $s.requests = s'.requests \setminus \{j\}$
  $s.promises\text{-}to[j] = t$

– $ready_i$
  P: $s'.region = \mathrm{T}$
  $s'.pending = \emptyset$
  $\min(s'.promises\text{-}to) \geq lb\text{-}time(s')$
  $\forall j \in s'.try\text{-}set,$
  $s'.promises\text{-}from[j] < (\infty, n)$
  E: $s.mctime = lb\text{-}time(s')$
  $s.region = \mathrm{R}$

– $send(\text{multicast}(m \in \mathscr{S}, t \in \mathscr{T}), i, j \in \mathscr{I})$
  P: $s'.region = \mathrm{M}$
  $s'.promises\text{-}from[j] < (\infty, n)$
  $t = s'.mctime$
  **if** $(j \in s'.try\text{-}set)$ **then**
  $m = s'.text$
  **else** $m = \varepsilon$
  E: $s.requested = s'.requested \setminus \{j\}$
  $s.promises\text{-}from[j] = (\infty, n)$

– $multicast\text{-}rcv_i(m)$
  P: $s'.region \in \{\mathrm{P}, \mathrm{T}\}$
  $(m, t) = \min(s'.pending)$
  $t < \min(s'.promises\text{-}to)$
  E: $s.pending = s'.pending \setminus \{(m, t)\}$
  $s.clock = t$
  **if** $s'.region = \mathrm{T}$ **then** $s.region = \mathrm{W}$

– $done_i$
  P: $s'.region = \mathrm{M}$
  $s'.requested = \emptyset$
  E: $s.need\text{-}set = \emptyset$
  $s.region = \mathrm{P}$

– $send(\text{adv-promise}(t \in \mathscr{T}), i, j \in \mathscr{I})$
  P: $s.region \in \{\mathrm{T}, \mathrm{W}\}$
  $\forall k \in s'.try\text{-}set,$
  $s'.promises\text{-}from[k] < (\infty, n)$
  $s'.promises\text{-}from[j] < lb\text{-}time(s')$
  $t = lb\text{-}time(s')$
  E: $s.promises\text{-}from[j] = lb\text{-}time(s')$

**Fig. 3.** Transition relation for $p_i$

$try_i(S)$ action causes $p_i$ to remember $S$ as its *try-set*, insert the elements of $S$ into its *need-set*, and enter its trying region (T). In region T, $p_i$ begins to send "req-promise" messages to each member of *need-set*, keeping track, in the component *requested*, of those requests already made in order to avoid sending duplicate requests. Each process $p_i$ receiving a "req-promise" message eventually responds by sending back a "promise" message with an associated logical time $t$.[7] The promise means that $p_j$ will not perform or deliver any multicasts with a time greater than $t$ until $p_i$ either relinquishes the promise (by sending a "multicast" message to $p_j$) or advances

[7] Note that $p_i$ sends "req-promise" messages to itself in order to simplify the presentation of the algorithm. A simple optimization would be to eliminate these messages, as well as the "promise" messages that $p_i$ sends to itself in response

the promise (by sending an "adv-promise" message with the later time). One may think of a promise as a roadblock that $p_j$ 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 advanced by $p_i$.

In order to ensure that progress is made, we would like each process to grant its promises with logical times that are "far enough in the future" to not impede its own progress. Therefore, for each $j$ in $\mathscr{I}$, it is useful to have a function *lb-time* that maps the states of $p_j$ to logical times. One may think of *lb-time* as a *lower bound* on the logical time that $p_i$ could assign to its next multicast. If $s$ is a state of $p_j$, we define *lb-time* for $p_j$ in state $s$ to be the least logical time having process-id $j$ such that

$$lb\text{-}time_j(s) \geq \max(s.clock,\ s.mctime,\ \max(s.promises\text{-}from)).$$

The subscript and/or argument of the *lb-time* function are sometimes omitted when their values are clear from context. We use the *lb-time* function to assign times to promises as follows: The time associated with a promise granted by $p_j$ from state $s$ is chosen by $p_j$ to be greater than the greatest logical time associated with any message in its *s.pending*, and also to be greater than *lb-time*(*s*).

Each process keeps track of both the times for promises it has granted to other processes (in the *promises-to* array) and the times for promises it has received from other processes (in the *promises-from* array). After receiving a promise from each process $p_j$ in its *try-set*, $p_i$ can issue a $ready_i$ action and assign *mctime* to the current value of *lb-time*, provided that (1) $p_i$'s *pending* set is empty, and (2) all promises $p_i$ has granted with times lower than *lb-time* have either been relinquished or advanced past *lb-time*. The second condition is present to ensure that $u_i$ receives no multicast messages with logical times less than $t$ after $p_i$ decides to send its multicast. Note that once *mctime* is assigned in a $ready_i$ action, it remains fixed for all further processing of $u_i$'s current multicast. Specifically, any further change in the *lb-time* leaves the *mctime* unaffected.

When a $ready_i$ action occurs, $u_i$ can no longer back out from sending a multicast. The $ready_i$ action leaves $p_i$ in the ready region (R), where it waits for $u_i$ to respond with a $multicast\text{-}send_i(m)$ action. When this action occurs, $p_i$ enters the multicast region (M) and records the desired text of the multicast in its *text* component. In region M, $p_i$ sends "multicast" messages to all processes $p_j$ from whom it holds promises. These messages have two purposes. First, they communicate the *text* and *mctime* of the multicast. Second, they relinquish the promises. If $p_i$ holds a promise from $p_j$ but $j$ is not in *try-set* (we will see shortly how this may happen), the text argument of the multicast message is set to $\varepsilon$, indicating that the promise should be relinquished but that no multicast should be delivered to $u_j$. After $p_i$ has relinquished all the promises it requested, it may issue a $done_i$ action and return to its passive region.

When a process $p_j$ receives a multicast$(m, t)$ message from $p_i$, it notes that its promise to $p_i$ has been relinquished, and, if $m \neq \varepsilon$, inserts the pair $(m, t)$ into its pending set. The message $m$ is eventually delivered to $u_j$ in a $multicast\text{-}rcv_j(m)$ action when $t$ is the least time among the times in $p_j$'s pending set and $p_j$ has no outstanding promises with times less than $t$. These conditions are necessary to ensure that any later $(m', t')$ pair received by $p_j$ will have $t' > t$ so that multicast messages are delivered in logical time order.

So far in this discussion, we have ignored the fact that many multicasts may be proceeding concurrently. Two complications arise as a result of this concurrency. The first relates to the delivery of a multicast message to a user while that user is itself waiting to send a multicast, and the second results from the need to break deadlock situations that result from the granting of promises. We now consider each of these complications in turn.

If $p_i$ is in region T and issues a $multicast\text{-}rcv_i(m)$ action, it enters the waiting region (W) where it waits for a response from $u_i$. Process $u_i$, on the basis of the new message $m$, may decide either to continue trying to perform a multicast or to back out. In case of the former, $u_i$ issues a $try_i(S')$ action, where $S'$ is not necessarily the same as $S$[8]. This $try_i$ action is treated just as before. If $u_i$ decides to back out, it instead issues a $backout_i$ action, causing $p_i$'s *try-set* to become empty and causing $p_i$ to enter region M, where it proceeds to relinquish its promises as usual.

In the course of concurrent scheduling of multicasts, deadlock situations may arise from the granting of promises. 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 $p_i$ (including the one received from $p_j$) are less than some promise received by $p_j$. Then $p_i$'s *lb-time* is less than that of $p_j$. If $p_i$ has granted $p_j$ a promise less than $p_i$'s own *lb-time*, then neither can perform a multicast before the other because each must wait for the other to relinquish its promises. Such deadlock situations are avoided by *promise advancement* as follows. 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 not to block unnecessarily the computation of each process $p_j$ from which $p_i$ has received a promise, $p_i$ may send $p_j$ an "adv-promise" message, informing it of $p_i$'s current *lb-time*. Upon receiving in "adv-promise" message from $p_i$, $p_j$ notes that its promise to $p_i$ has been advanced. This may permit $p_j$ to deliver additional multicast messages from its pending set and/or proceed with its own multicast. In the liveness proof, we will show that these "adv-promise" messages are sufficient to guarantee progress.

In studying the algorithm, one will notice a great deal of nondeterminism in the ordering of events. For example, we have not specific the order in which promises are requested from different processes. As a result of this nondeterminism, the correctness proof of the algorithm is more general, covering many possible implementations of the algorithm.

[8] Recall that our liveness condition says that even if $u_i$ "changes its mind" about the particular multicast it wishes to send, as long as it continually has *some* multicast that it wishes to send, eventually it must be permitted to do so. The ability to change the set of recipients explains how $p_i$ may hold promises from processes not named in its *try-set*

## 5 Proof of correctness

Let module $P$ be the composition of all automata $p_i$, $i \in \mathscr{I}$. In this section, we show that module $P$ solves schedule module $M$, which implies that the logically synchronous multicast protocol is correct. The organization of the correctness proof closely follows the definition of schedule module $M$. Clearly, $\operatorname{sig}(P) = \operatorname{sig}(M)$. 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. Throughout the proof, we use subscripts to distinguish the state components of the different automata in $P$. For example, $region_i$ is the *region* variable in the local state of automaton $p_i$.

### 5.1 Safety proof

As we have said, the safety proof consists of showing that all executions of $P$ satisfy conditions (1) and (2) of schedule module $M$. We start by proving condition (1), that $P$ preserve user well-formedness for all $i \in \mathscr{I}$. Following this, we state some properties of well-formed executions that will be used in the proof of condition (2), as well as in the liveness proof. A key part of proving condition (2) is showing the existence of a proper correspondence relation $\mathscr{C}$ on the *multicast-send* and *multicast-rcv* events in any execution $\alpha$ of $P$, and also showing the existence of a proper total order on the *multicast-send* events in $\alpha$. To accomplish this, we exhibit particular constructions that produce a correspondence relation $\mathscr{C}_\alpha$ and an ordering $\prec_\alpha$ for any execution $\alpha$ of $P$. We then show that $\prec_\alpha$ is indeed a total order and finally that condition (2) is satisfied. We prove the three parts of condition (2) with the help of several intermediate lemmas.

We now turn to the proof of condition (1). The following relationship between the state of $p_i$ and the definition of $r(i, \alpha)$ can be shown by induction on the length of $\alpha$.

**Lemma 1.** *Let $\alpha$ be a prefix of an execution of $P$ that is user well-formed for all $i \in \mathscr{I}$, and let $s$ be the last state of $\alpha$. Then for all $i \in \mathscr{I}$, $s.region_i = r(i, \alpha)$.*

From the above lemma, it follows that module $P$ satisfies condition (1) of schedule module $M$. Again, the proof is a simple induction on the length of the execution.

**Theorem 2.** *Module $P$ preserves user well-formedness for $i$, for all $i \in \mathscr{I}$.*

We know that module $P$ preserves user well-formedness for all $i \in \mathscr{I}$. Furthermore, since no *rcv* action is an output of $P$, it is not possible for $P$ to violate network well-formedness. Therefore, in the remaining proofs we can restrict our attention to well-formed executions only. This motivates the following convenient definition. Let $\alpha$ be an execution of $P$. We say that $\alpha$ is *admissible* iff $\alpha$ is user well-formed for every $i \in \mathscr{I}$ and $\alpha$ is network well-formed. The following lemma states some properties of admissible executions that will be used throughout the proof.

**Lemma 3.** *Let $\alpha$ be an admissible execution of $P$. For any $i \in \mathscr{I}$, let $\alpha'$ be a subexecution of $P$ between two successive $done_i$ events, (or between the beginning of $\alpha$ and the first $done_i$ event). Then $\forall j \in \mathscr{I}$, if $\alpha'$ contains an event having* any *of the following forms, then it contains* exactly one *event of each form 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(multicast($m$, $t''$), $i$, $j$), where $m \in \mathscr{S}$, $t$, $t'' \in \mathscr{T}$. Furthermore, any events of the form send(adv-promise($t'$), $i$, $j$), $t' \in \mathscr{T}$, occurring in $\alpha'$ must appear between the last two of the above events.*

*Proof.* The proof is by induction, assuming that the conditions hold for $i$ in the prefix of $\alpha$ up to the beginning of $\alpha'$.

First we show that no two *send*(req-promise, $i$, $j$) events can occur in $\alpha'$. The action $\pi_1 = send$(req-promise, $i$, $j$) is only enabled when $region_i = \mathrm{T}$ and $j \notin requested_i$. When the action occurs, it results in $j \in requested_i$. Elements may be deleted from the set $requested_i$ only while $region_i = \mathrm{M}$. Therefore, another action *send*(req-promise, $i$, $j$) cannot occur after $\pi_1$ until $p_i$ passes through some state in which $region_i = \mathrm{M}$ and then reaches a state in which $region_i = \mathrm{T}$. By Lemma 1 and the definition of user well-formedness, this cannot happen without an intervening $done_i$.

Next, we show that if $\pi_1 = send$(req-promise, $i$, $j$) occurs in $\alpha$, then the next $done_i$ event after $\pi_1$ must be preceded by $\pi_5 = send$(multicast($m$, $t''$), $i$, $j$). The action $\pi_1$ has as an effect that $j \in requested_i$, and $done_i$ has as a precondition that $requested_i$ is empty. Therefore, since $\pi_5$ is the only action that can remove $j$ from $requested_i$, it must occur between $\pi_1$ and $done_i$.

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 that occur before $\alpha'$ have their corresponding receives occur before $\alpha'$. Therefore, by network well-formedness, $\pi_2 = rcv$(req-promise, $i$, $j$) cannot occur before $\pi_1$, and only one $\pi_2$ 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 $\pi_2$ is the only action that can cause $i \in requests_j$, it must precede $\pi_3$. Again, by network well-formedness and the induction hypothesis, we know that $\pi_3$ must precede $\pi_4 = rcv$(promise($t$), $j$, $i$). The action $\pi_5 = send$(multicast($m$, $t''$), $i$, $j$) has as a precondition that $promises\text{-}from_i\,[j] < (\infty, n)$. Since $\pi_5$ has as an effect that $promises\text{-}from_i\,[j] = (\infty, n)$, and since $\pi_4$ is the only action that can cause $promises\text{-}from_i\,[j] < (\infty, n)$, we know by the induction hypothesis that $promises\text{-}from_i\,[j] = (\infty, n)$ at the beginning of $\alpha'$. Therefore, $\pi_4$ must precede $\pi_5$.

Since *send*(adv-promise($t'$), $i$, $j$) has as a precondition that $promises\text{-}from_i\,[j] < (\infty, n)$, we know that it cannot occur before $\pi_4$ or after $\pi_5$. $\square$

In the remainder of the proof, we often use the above lemma to show the existence or nonexistence of particular events in a portion of an execution.

Conditions (2) and (3) of schedule module $M$ refer to the existence of a correspondence relation and a total order. In completing the proof, it is helpful to fix particular constructions for these as follows. Let $\alpha$ be an execution of $P$. For all $i\in\mathscr{I}$, if $\pi$ is a *multicast-send*$_i$ event occurring in $\alpha$ and $s$ is the state immediately preceding $\pi$, then we define $time(\pi, \alpha)$ to be $s.mctime_i$. Similarly, if $\pi$ is a *multicast-rcv*$_i$ event occurring in $\alpha$ and $s$ is the state immediately following $\pi$, then we define $time(\pi, \alpha)$ to be $s.clock_i$. We fix the correspondence relation $\mathscr{C}_\alpha$ as follows: For all $i, j\in\mathscr{I}$ and for all $m\in\mathscr{S}$, events $\pi_1 = multicast\text{-}send_i(m)$ and $\pi_2 = multicast\text{-}rcv_j(m)$ correspond in $\alpha$ iff $time(\pi_1, \alpha) = time(\pi_2, \alpha)$. We fix $\prec_\alpha$ to be the ordering as follows: For all $\pi_1, \pi_2$ *multicast-send* actions in $\alpha$, $\pi_1 \prec_\alpha \pi_2$ iff $time(\pi_1, \alpha) < time(\pi_2, \alpha)$.

Before proceeding with the three parts of condition (2), we must first show that $\prec_\alpha$ is indeed a total order on the *multicast-send* events. Recall that the construction of $\prec_\alpha$ is based upon assigning logical times to each *multicast-send*$_i$ event according to the value of $mctime_i$ in the preceding state. In the next lemma, we show that the state component $mctime_i$ is nondecreasing.

**Lemma 4.** *Let $\alpha$ be an admissible execution of $P$. Then for all $i\in\mathscr{I}$, if state $s'$ precedes state $s$ in $\alpha$, then $s'.mctime_i \le s.mctime_i$.*

*Proof.* The actions $ready_i$ are the only actions that modify $mctime_i$. These actions set $s.mctime_i$ to the value of $lb\text{-}time_i(s')$, which is no less than $s'.mctime_i$ by definition. □

With this lemma, we can now show that each multicast is assigned a unique logical time by the protocol.

**Lemma 5.** *Let $\alpha$ be an admissible execution of $P$. Let $\pi = multicast\text{-}send_i(m)$ and $\pi' = multicast\text{-}send_j(m')$ be two events in $\alpha$. Then $time(\pi, \alpha) \ne time(\pi', \alpha)$.*

*Proof.* There are two cases, depending on whether or not $\pi$ and $\pi'$ are outputs of different user processes. If $i \ne j$, then we know trivially that $time(\pi, \alpha) \ne time(\pi', \alpha)$ because they differ in the process-id. (The state component $mctime_i$ is assigned only to values in the range of $lb\text{-}time_i$, and these values have $i$ as the process-id by definition).

If $i=j$, then assume, without loss of generality, that $\pi'$ precedes $\pi$ in $\alpha$. From the definition of user well-formedness, we know that at least one $ready_i$ action occurs between $\pi'$ and $\pi$. Let $s'$ be the state from which the last such $ready_i$ action occurs, and let $s$ be the resulting state. We know from Lemma 4 that $s'.mctime_i$ is no less than the value of $mctime_i$ in the state after $\pi'$. Therefore, if we can show that $s'.mctime_i < s.mctime_i$, then we will have proven that $time(\pi', \alpha) < time(\pi, \alpha)$. By the precondition of $ready_i$, we know that in state $s'$, $p_i$ must hold a promise from itself for some logical time $t$. By Lemma 3 and user well-formedness for $i$, we know that $p_i$'s promise to itself is sent (and received) between the last preceding $done_i$ action and state $s'$. Also by user well-formedness, we know that no $ready_i$ action occurs between this $done_i$ action and state $s'$, so the value of $mctime_i$ is constant over that execution interval. Whenever $p_i$ sends a promise, the promise is assigned a time strictly greater than $p_i$'s own *lb-time*, which is, by definition, at least as large as its own *mctime*. Therefore, $t > s'.mctime$. Since $p_i$ holds a promise for time $t$ in state $s'$, we know that $lb\text{-}time_i(s') \ge t$. Therefore, since the $ready_i$ action assigns $mctime_i$ to the value of $lb\text{-}time_i$, we know that $s.mctime_i > s' \cdot mctime_i$. □

This immediately implies the desired result that the construction of $\prec$ produces a total order on the *multicast-send* events:

**Corollary 6.** *Let $\alpha$ be an admissible execution of $P$. Then $\prec_\alpha$ is a total order on the multicast-send events in $\alpha$.*

*Proof.* Immediate from Lemma 5 and the definition of $\prec_\alpha$. □

Having shown that $\prec_\alpha$ is a total order, we can turn to the main task of proving condition (2) of schedule module $M$. We begin with condition (2a).

**Theorem 7.** *Let $\alpha$ be an admissible execution of $P$. Then $\mathscr{C}_\alpha$ is a proper correspondence relation for $\alpha$.*

*Proof.* Let $\pi = multicast\text{-}send_i(m)$ be an event in $\alpha$, and let $try_i(S)$ be the last preceding $try_i$ action. By Lemma 5, we know that $\pi$ is assigned a unique logical time $t$ by the protocol. By the definition of $p_i$ and specifically the preconditions of the $send$(multicast$(m, t), i, j)$ action, we know that at most one $send$(multicast$(m \ne \varepsilon, t), i, j)$ action occurs in $\alpha$ for each $j\in S$ (and that none occurs for $j \notin S$). By network well-formedness, we know that at most one $rcv$(multicast$(m, t), i, j)$ occurs in $\alpha$ for each of these sends. So $(m, t)$ is added to $pending_i$ at most once in $\alpha$, for each $j\in S$ (and never for $j\notin S$). Therefore, by the definition of *multicast-rcv*, at most one *multicast-rcv*$_j(m)$ action corresponds to $\pi$ for each $j\in S$, and no such actions correspond to $\pi$ for $j\notin S$. This proves that $\mathscr{C}_\alpha$ satisfies properties 1 and 3 of the definition of a proper correspondence relation.

We now show property 2. By the construction of $\mathscr{C}_\alpha$, each *multicast-rcv* has an associated logical time and corresponds only to those *multicast-send* actions assigned this time. By Lemma 5, each *multicast-send* has a unique logical time, so each *multicast-rcv* can correspond to at most one *multicast-send*. It remains to be shown that each *multicast-rcv* has at least one corresponding *multicast-send*. Let $s'$ be the state from which a *multicast-rcv*$_j(m)$ action occurs and let $s$ be the resulting state. Then by the definition of that action, it must be that $(m, t)\in s'.pending_j$ and $s.clock_i = t$. Therefore, a $rcv$(multicast$(m, t), i, j)$ must have occurred prior to $s'$. By network well-formedness, this event must have been preceded by a $send$(multicast$(m, t), i, j)$, which could only have been enabled as a result of a *multicast-send*$_i(m)$ action with an assigned logical time of $t$. This is the desired corresponding action. □

The next part of the proof is to show that $\prec_\alpha$ is a proper total order for $\mathscr{C}_\alpha$ and $\alpha$. In order to accomplish this, we first prove a lemma that state 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". This is used to prove a second lemma, that the *clock* component of a process is nondecreasing. This will enable us to show the desired property of $\prec_\alpha$.

**Lemma 8.** *Let* $\alpha$ *be an admissible execution of P. Then for all* $i, j \in \mathscr{I}$, *the following properties hold for all states s in* $\alpha$.

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

*Proof.* Each property is proved by a separate induction on the length of $\alpha$[9].

*Property* (1). If $s$ is an initial state, then for all $i$, $j \in \mathscr{I}$, $i \notin s.requests_j$, so the statement holds vacuously. The only action that can falsify $s.promises\text{-}to_j[i] = (\infty, n)$ is *send*(promise($t$), $i$, $j$), but this action removes $i$ from $s.requests_j$. The only action that 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 $\alpha'$. Suppose (for contradiction) that $s.promises\text{-}to_j[i] < (\infty, n)$. This can only be true if there exists some $\pi'$, either a *send*(promise($t$), $j$, $i$) or a *rcv*(adv-promise($t$), $i$, $j$), in $\alpha'$ such that no *rcv*(multicast($m$, $t'$), $i$, $j$) occurs between $\pi'$ and $\pi$. However, by Lemma 3, every *send*(promise($t$), $j$, $i$) or *send*(adv-promise($t$), $i$, $j$) must be followed by a *send*(multicast($m$, $t'$), $i$, $j$) before the next *send*(req-promise, $i$, $j$) occurs. So by network well-formedness, *rcv*(multicast($m$, $t'$), $i$, $j$) must occur between $\pi'$ and $\pi$, giving us a contradiction.

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

1. If $\pi = send$(promise($t$), $j$, $i$), then by Property (1) and the preconditions on $\pi$, $s'.promises\text{-}to_j[i] = (\infty, n)$. Therefore, $promises\text{-}to_j[i]$ is not increased by $\pi$.
2. If $\pi = rcv$(promise($t$), $j$, $i$), then $s.promises\text{-}from_i[j] = t$. By network well-formedness, $\pi' = send$(promise($t$), $j$, $i$) must occur earlier in $\alpha'$, leaving $promises\text{-}to_j[i] = t$. The only possible events that could occur between $\pi'$ and $\pi$ to make $s.promises\text{-}to_j[i] \neq t$ are *rcv*(adv-promise($t'$), $i$, $j$) or *rcv*(multicast($m$, $t'$), $i$, $j$). By Lemma 3, we know that $\pi'$ must occur before $\pi$ such that no *send*(adv-promise($t'$), $i$, $j$) or *send*(multicast($m$, $t'$), $i$, $j$) occurs between $\pi'$ and $\pi$. By the same lemma, we know that a $\pi'' = rcv$(req-promise, $i$, $j$) occurs before $\pi'$ such that no *send*(adv-promise($t'$), $i$, $j$) or *send*(multicast($m$, $t'$), $i$, $j$) occurs between $\pi''$ and $\pi'$. Hence, by network well-formedness, no *rcv*(adv-promise($t'$), $i$, $j$) or *rcv*(multicast($m$, $t'$), $i$, $j$) occurs between $\pi'$ and $\pi$.
3. If $\pi = rcv$(adv-promise($t'$), $i$, $j$), then $s.promises\text{-}to_j[i] = t'$. By Lemma 3 and network well-formedness, the corresponding *send*(adv-promise($t'$), $i$, $j$) must follow a $\pi' = send$(promise($t$), $j$, $i$) such that no *rcv*(multicast($m$, $t'''$), $i$, $j$) occurs between them. By the preconditions of *send*(adv-promise($t'$), $i$, $j$), $t' > t$, and that action results in $promises\text{-}from_i[j] = t'$. Furthermore, any other *send*(adv-promise($t''$), $i$, $j$) occurring in $\alpha'$ after *send*(adv-promise($t'$), $i$, $j$) must have $t'' > t'$. Therefore, the property holds.
4. If $\pi = rcv$(multicast($m$, $t$), $i$, $j$), then $s.promises\text{-}to_j[i] = (\infty, n)$. By network well-formedness, $\pi$ must be preceded by $\pi' = send$(multicast($m$, $t$), $i$, $j$), resulting in $promises\text{-}from_i[j] = (\infty, n)$. The only action that can decrease $promises\text{-}from_i[j]$ is a *rcv*(promise($t'$), $j$, $i$). But by Lemma 3, any *rcv*(promise($t'$), $j$, $i$) occurring between $\pi'$ and $\pi$ must be preceded in that interval by a *send*(req-promise, $i$, $j$) and a *rcv*(req-promise, $i$, $j$). But this violates network well-formedness (S4), so no *rcv*(promise($t'$), $j$, $i$) occurs between $\pi'$ and $\pi$. Therefore $s.promises\text{-}from_i[j] = (\infty, n)$.

*Property* (3). The base case, $\alpha$ a start state, holds since $clock_j = (0, j)$ and $promises\text{-}to_j[i] = (\infty, n)$ for all $i \in \mathscr{I}$. Now, consider those actions that can potentially increase $clock_j$ or decrease $promises\text{-}to_j[i]$. These are $multicast\text{-}rcv_j$, *send*(promise($t$), $j$, $i$). and *rcv*(adv-promise($t$), $i$, $j$). By definition, the action $multicast\text{-}rcv_j$ sets clock to a value $t$, such that $\forall i \in \mathscr{I}$, $promises\text{-}to_j[i] > t$. The action *send*(promise($t$), $j$, $i$) sets $promises\text{-}to_j[i] = t$ and is enabled only if $t > lb\text{-}time_j$, which is at least $clock_j$ by definition. Finally, the action *rcv*(adv-promise($t$), $i$, $j$) sets $promises\text{-}to_j[i] = t$. To show that $t > clock_j$, we note that *send*(adv-promise($t$), $i$, $j$) is enabled at $p_i$ only if $promises\text{-}from_i[j] < t$. Therefore, by Property (2), $t > promises\text{-}to_j[i]$ when *send*(adv-promise($t$), $i$, $j$) occurs. And therefore, $t > promises\text{-}to_j[i]$ when *rcv*(adv-promise($t$), $i$, $j$) occurs, since Lemma 3 and network well-formedness tell us that neither a *rcv*(multicast($m$, $t'$), $i$, $j$) 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). The base case, $\alpha$ a start state, holds since $region_i = \text{P}$. Let $\alpha = \alpha' \pi s$, where the property holds after $\alpha'$. There are two cases.

We first consider the case in which $p_i$ enters region R, and subsequently enters region M. In this case, $\pi = ready_i$, so the property holds by the preconditions and effects of $ready_i$. In that action, *lb-time* and *mctime* are made equal, and we note that *mctime* remains unchanged until after $p_i$ exits region M. We also observe that by user well-formedness for $i$, no $try_i$ actions can occur from regions R or M, so $try\text{-}set_i$ is fixed in R and M. By Lemma 3, no new promises from members of *try-set* are received by $p_i$ while in R or M, since those promises have already been received (by precondition of $ready_i$). Therefore, to show that the property holds after all extensions of $\alpha$ in which $p_i$ remains in R or M, we need only show that for all $j \in try\text{-}set$, if $promises\text{-}from_i[j]$ is

[9] This is in contrast to proofs in which the inductive hypothesis includes all of the invariants

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

For the second case, $p_i$ does not enter M from region R. In this case, $\pi$ must be $backout_i$, by user well-formedness for $i$. Therefore, by the effects clause of that action, $try\text{-}set_i = \emptyset$, so the property holds vacuously until the next $done_i$.

*Property* (5). 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 $multicast\text{-}rcv_j$, which removes the element from $pending_j$ having the least time, and sets $clock_j$ to that time. By Lemma 5, no two *multicast-send* actions are assigned the same logical time. So by Lemma 3, at most one *send*(multicast($m$, $t$), $i, j$) occurs for a given time $t$. And by network well-formedness, at most one *rcv*(multicast($m$, $t$), $i, j$) occurs. Therefore, no two items in $pending_j$ have the same logical time. So by the induction hypothesis, the property holds.

The action $\pi = rcv(\text{multicast}(m \neq \varepsilon, t), i, j)$, for some $i \in \mathscr{I}$, is the only action that can add elements to $pending_j$. Let $s''$ be the state from which the corresponding *send*(multicast($m$, $t$), $i, j$) occurs. Since $m \neq \varepsilon$ implies that $j \in s''.try\text{-}set_i$, we know from Property (4) that $s''.promises\text{-}from_i[j] \leq t = s''.mctime_i$. Therefore, by Property (2), $s''.promises\text{-}to_j[i] \leq t$. By Lemma 3 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 $s'$ that could cause $promises\text{-}to_j[i]$ to increase past $t$. Therefore $s'.promises\text{-}to_j[i] \leq t$. So, by Property (3), $s'.clock_j < t$. When $\pi$ occurs, $(m, t)$ is added to $pending_j$, so Property (5) holds in state $s$. □

We now show that the *clock* state component is nondecreasing.

**Lemma 9.** *Let $\alpha$ be an admissible execution of P. Then for all $i \in \mathscr{I}$, if state $s'$ precedes state $s$ in $\alpha$, then $s'.clock_i \leq s.clock_i$.*

*Proof.* Consider the actions $multicast\text{-}rcv_i$, which are the only actions in which $clock_i$ can be modified. Whenever a $multicast\text{-}rcv_i$ action is enabled, $pending_i$ is nonempty. By definition, a $multicast\text{-}rcv_i$ action results in $clock_i$ being set to the minimum logical time in $pending_i$. By Property (5) of Lemma 8, $clock_i$ is less than the minimum logical time in $pending_i$, provided $pending_i$ is nonempty. Therefore, whenever $clock_i$ is modified, its value is increased. □

We can now prove property (2b) of schedule module $M$.

**Theorem 10.** *Let $\alpha$ be an admissible execution of P. Then $\prec_\alpha$ is a proper total order for $\mathscr{C}_\alpha$ and $\alpha$.*

*Proof.* We need to show that $\forall i, j, k \in \mathscr{I}$ and $\forall m, m' \in \mathscr{S}$, if $\pi = multicast\text{-}send_i(m)$ and $\pi' = multicast\text{-}send_j(m')$ occur in $\alpha$ with corresponding receives $\hat{\pi} = multicast\text{-}rcv_k(m)$ and $\hat{\pi}' = multicast\text{-}rcv_k(m')$, and if $\prec_\alpha$ orders $\pi'$ before $\pi$, then $\hat{\pi}'$ occurs before $\hat{\pi}$.

From Lemma 5, we know that $\pi$ and $\pi'$ have associated unique logical times. Let these be $t$ and $t'$, respectively. Since $\prec_\alpha$ orders $\pi'$ before $\pi$, we know that that $t > t'$. Furthermore, by the definition of $\mathscr{C}_\alpha$, we know that $clock_k = t$ in the state immediately after $\hat{\pi}$ and that $clock_k = t'$ in the state immediately after $\hat{\pi}'$. Lemma 9 tells us that $clock_k$ is nondecreasing. Therefore, $\hat{\pi}'$ must occur before $\hat{\pi}$. □

Finally, we prove property (2c) to complete the safety proof.

**Theorem 11.** *Let $\alpha$ be an admissible execution of P with correspondence relation $\mathscr{C}_\alpha$. Then $\forall j \in \mathscr{I}$ and $\forall m, m' \in \mathscr{S}$, if $\pi = multicast\text{-}send_i(m)$ occurs in $\alpha$, then no $multicast\text{-}rcv_i(m')$ occurs between $\pi$ and the corresponding $\hat{\pi} = multicast\text{-}rcv_i(m)$.*

*Proof.* Consider the state $s$ from which $\pi$ occurs, let $\alpha'$ be the prefix of $\alpha$ ending in state $s$, and let $t = s.mctime_i \equiv time(\pi, \alpha')$. We know, from user well-formedness for $i$, that $r(i, \alpha') = \text{R}$. Consider the last action $ready_i$ occurring in $\alpha'$, 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 user well-formedness for $i$, that $region_i = \text{R}$ at all states between $s'$ and $s$.

Suppose (for contradiction) that a $multicast\text{-}rcv_i(m')$ occurs between $\pi$ and $\hat{\pi}$. From the definition of $ready_i$, we know that $s'.pending_i = \emptyset$. Therefore, the only way for the $multicast\text{-}rcv_i(m')$ to occur between $s'$ and $\hat{\pi}$ is for a *rcv*(multicast($m' \neq \varepsilon$, $t''$), $j, i$) with $t'' < t$ to occur first in that interval. By the preconditions of $ready_i$, $s'.promises\text{-}to_i[j] \geq lb\text{-}time(s'|i) = t$, for all $j \in \mathscr{I}$. Furthermore, any later *send*(promise($t'$), $i, j$) must have $t' > lb\text{-}time_i$, which is greater than $mctime_i$ in every state by definition. From Lemma 4, we know that $mctime_i$ is nondecreasing, so $mctime_i \geq t$ in all states after $s'$. Therefore, by Properties (2) and (4) of Lemma 8, no *send*(multicast($m' \neq \varepsilon$, $t''$), $j, i$) with $t'' < t$ can occur after $s'$. (We ignore *send*(multicast($\varepsilon$, $t''$), $j, i$) actions here because a *rcv*(multicast($\varepsilon$, $t$), $j, i$) action does not cause an element to be inserted into the *pending* set). So the only way for a *rcv*(multicast($m'$, $t''$), $j, i$) with $t'' < t$ to occur between $s'$ and $\hat{\pi}$ is for its corresponding send to occur before $s'$. If this is the case, then by Properties (2) and (4) of Lemma 8, $s'.promises\text{-}to_i[j] \leq t''$. But this violates the precondition for the $ready_i$ action that occurs from state $s'$. □

### 5.2 Liveness proof

The liveness proof consists of showing that executions of $P$ satisfy condition (3) of schedule module $M$. We prove the two parts of condition (3) in order. Since the protocol is required to make progress only if the user processes and the network satisfy their liveness properties, we will restrict our attention to only those executions in which the environment is live. This motivates the following definition. Let $\alpha$ be a fair execution of $P$. We say that $\alpha$ is *well-behaved* iff $\alpha|U_i \in scheds(U_i)$ for all $i \in \mathcal{I}$ and $\alpha|N \in scheds(N)$. Note that every well-behaved execution is an admissible execution, by the definitions of $U_i$ and $N$, and the fact that $P$ preserves user well-formedness for all $i \in \mathcal{I}$.

Before proving condition (3a), we prove four intermediate lemmas. The following lemma states that if a promise is requested, then eventually it is granted.

**Lemma 12.** *Let $\alpha$ be a well-behaved execution of $P$. If event $\pi = send$(req-promise, $i$, $j$) occurs in $\alpha$ then a later $rcv$(promise$(t)$, $j$, $i$) occurs in $\alpha$ for some $t \in \mathcal{T}$.*

*Proof.* By the definition of $scheds(N)$, a $\pi' = rcv$(req-promise, $i, j$) occurs in $\alpha$ after $\pi$. By the transition relation for $p_j$, $i \in requests_j$ in the state after $\pi'$. Only a $send$(promise$(t)$, $j$, $i$) action can cause $i \notin requests_j$. Therefore, a $send$(promise$(t)$, $j$, $i$) action is enabled in all states after $\pi'$ until one occurs. Since $\alpha$ is a fair execution and $send$(promise$(t)$, $j$, $i$). actions are in their own class of the partition, such an action eventually occurs. The definition of $scheds(N)$ tells us that a corresponding $rcv$(promise$(t)$, $j$, $i$) occurs later in $\alpha$. □

The following simple lemma states that if a $try_i$ action occurs, then eventually either $need\text{-}set_i$ becomes fixed, or else a later $ready_i$ or $backout_i$ action occurs.

**Lemma 13.** *Let $\alpha$ be a well-behaved execution of $P$, and let $\alpha'$ be a suffix of $\alpha$ beginning with a $try_i$ action, for $i \in \mathcal{I}$. If no $backout_i$ or $ready_i$ action occurs in $\alpha'$ then there exists a state in $\alpha'$ after which $need\text{-}set_i$ is fixed.*

*Proof.* If no $backout_i$ or $ready_i$ action occurs in $\alpha'$, then from the definitions of $p_i$ and user well-formedness we know that no element is deleted from set $need\text{-}set_i$ in $\alpha'$. Therefore, since $need\text{-}set_i$ can contain at most $n$ elements, we know that there exists a state in $\alpha'$ after which $need\text{-}set_i$ is not changed. □

The next lemma states that a process can eventually accumulate promises from all processes named in its *need-set*. This fact will be useful in proving Lemma 15.

**Lemma 14.** *Let $\alpha$ be a well-behaved execution of $P$, and let $\alpha'$ be a suffix of $\alpha$ beginning with a $try_i$ action. If neither a $backout_i$ nor a $ready_i$ action occurs in $\alpha'$, then there must exist a point in $\alpha'$ after which the following condition holds for all states $s$: $\forall j \in s.need\text{-}set_i$, $s.promises\text{-}from_i[j] < (\infty, n)$.*

*Proof.* If no $backout_i$ or $ready_i$ action occurs in $\alpha'$ then by user well-formedness for $i$, $region_i \in \{\mathrm{T}, \mathrm{W}\}$ in all states of $\alpha'$. From Lemma 13, there exists a state $s'$ in $\alpha'$ after which $need\text{-}set_i$ is fixed. Let $\alpha''$ be the suffix of $\alpha'$ beginning with state $s'$. Then for each state $s''$ in $\alpha''$ and for each $j \in s'.need\text{-}set$, there are two possibilities: either (1) $j \notin s' \cdot requested_i$, and $send$(req-promise, $i$, $j$) is enabled or (2) $j \in s'.requested_i$ and $send$(req-promise, $i$, $j$) occurs before $s'$ (and after the last preceding $done_i$, if one occurs). In case (1), we know that a $send$(req-promise, $i$, $j$) must eventually occur since $\alpha$ is a fair execution and such actions form their own class of the partition. So, in either case, Lemma 12 tells us that a $rcv$(promise$(t)$, $j$, $i$) action must occur in $\alpha$ (after the last $done_i$ event, if one occurs). So eventually, $promises\text{-}from_i[j] < (\infty, n)$ for all $j \in need\text{-}set$. No action can occur at $p_i$ in region T or W to cause an entry in the $promises\text{-}from_i$ array to become $(\infty, n)$. Thus, we have the desired result. □

The final intermediate lemma states that if a process is attempting to perform a multicast, then eventually its *lb-time* will stop increasing or the process will perform a multicast.

**Lemma 15.** *Let $\alpha$ be a well-behaved execution of $P$, and let $\alpha'$ be a suffix of $\alpha$ beginning with a $try_i$ action. If neither a $backout_i$ nor a $ready_i$ action occurs in $\alpha'$, then there exist a logical time $t \in \mathcal{T}$ and a state $s$ in $\alpha'$ such that $lb\text{-}time_i = t$ in all states after $s$.*

*Proof.* If no $backout_i$ or $ready_i$ action occurs in $\alpha'$, then from Lemmas 13 and 14 we know that there exists a state $s$ in $\alpha'$ after which $need\text{-}set_i$ is fixed and $p_i$ holds promises from all processes named in $need\text{-}set_i$. Let $t = lb\text{-}time_i(s)$. In order to show that $lb\text{-}time_i$ cannot grow past $t$ in $\alpha'$, we need to show that no new promises arrive at $p_i$, that $p_i$ does not advance any promises past $t$, and that $clock_i$ and $mctime_i$ do not increase past $t$. Clearly, since $need\text{-}set_i$ is fixed and $p_i$ holds promises from each process named in $need\text{-}set_i$, no new promises are requested and no new promises arrive. And by definition, $p_i$ never advances a promise beyond its current *lb-time.* Since $p_i$ holds a promise from itself (for a time $\leq t$), we know by Property (3) of Lemma 8 that $clock_i$ cannot grow past $t$. Finally, since $mctime_i$ is only modified by a $ready_i$ action, we know that this is fixed as well. □

The next two theorems correspond to Conditions (3a) and (3b) of schedule module $M$. In the first, we assume that there exists a set of blocked processes, and derive a contradiction by showing that the process with the least *lb-time* must eventually make progress. The promise advancement mechanism is crucial to this result, because it allows a process to discover that it is the one with the least *lb-time*. From the previous result, we know that only a finite number of these promise advancement messages are sufficient to ensure that progress is made.

**Theorem 16.** *Let $\alpha$ be a well-behaved execution of $P$. If a $try_i$ occurs in $\alpha$, then either a $backout_i$ or a $ready_i$ occurs later in $\alpha$.*

*Proof.* Suppose (for contradiction) that there exists a set $\mathcal{J} \subseteq \mathcal{I}$ such that $\forall j \in \mathcal{J}$, a $try_j$ occurs in $\alpha$ and no later $backout_j$ or $ready_j$ occurs in $\alpha$. From Lemmas 14 and

15, we know that there exists a suffix $\alpha'$ of $\alpha$ such that for all $j \in \mathscr{J}$,

1. *lb-time*$_j$ is fixed in $\alpha'$, and
2. for all states of $\alpha'$, $p_j$ holds a promise from every process and named in *try-set*$_j \subseteq$ *need-set*$_j$.

Let $i \in \mathscr{J}$ be the index of the process with the least *lb-time* in $\alpha'$, and let $t$ be this *lb-time*. To derive a contradiction, we wish to show that a *ready*$_i$ action occurs in $\alpha'$.

Given the preconditions on *ready*$_i$, there are only two ways in which the *ready*$_i$ action could not be enabled: Either (1) *promises-to*$_i[j] <$ *lb-time*$_i$ for some $j \in \mathscr{I}$, or (2) *pending*$_i$ is not empty. We consider these in order. By the preconditions on granting a promise, any new promises granted by $p_i$ in $\alpha'$ have logical times greater than $t$, so we need only consider promises granted before $\alpha'$. Each process $k \in \mathscr{I} \setminus \mathscr{J}$ makes progress (i.e., has a *backout*$_k$ or *ready*$_i$ action), and therefore reaches region $M$, where it eventually relinquishes every promise held. So, any promise that $p_i$ has granted to any process $p_k \in \mathscr{I} \setminus \mathscr{J}$ for a time less than $t$ must eventually be relinquished. We have already said that the remaining processes $p_j \in \mathscr{J}$ hold promises from all processes named in their *try-sets*. Therefore, since $\alpha$ is a fair execution, a *send*(adv-promise($t'$), $j$, $i$) occurs with $t'$ being the logical time at which *lb-time*$_j$ is fixed. By the definition of $N$, a corresponding *rcv*(adv-promise($t'$), $j$, $i$) occurs later in $\alpha$. Since $p_i$ has the least *lb-time* among processes named in $\mathscr{J}$, we know that $t' > t$ in all cases. Therefore, all promises that $p_i$ has granted to other processes for times less than $t$ are eventually relinquished or advanced past $t$. So, for all $j \in \mathscr{I}$, *promises-to*$_i[j] \geq$ *lb-time*$_i$. Therefore, by Property (5) of Lemma 8, nothing prevents *multicast-rcv*$_i$ actions from occurring to empty *pending*$_i$, since *lb-time*$_i \geq$ *clock*$_i$. Thus, since $\alpha$ is a fair execution, *ready*$_i$ eventually becomes enabled and must eventually occur. □

Finally, we show condition (3 b), that a multicast message is eventually delivered to all the destination processes.

**Theorem 17.** *Let $\alpha$ be a well-behaved execution of $P$. If a multicast-send$_i(m)$ occurs in $\alpha$ and try$_i(S)$ is the last preceding try$_i$ action in $\alpha$, then a multicast-rcv$_j(m)$ occurs later in $\alpha$ for each $j \in S$.*

*Proof.* If *multicast-send*$_i(m)$ occurs in $\alpha$, we know that a *ready*$_i$ must precede it, by user well-formedness for $i$. By the preconditions of *ready*$_i$, for all $j \in$ *try-set*$_i = S$, *promises-from*$_i[j] < (\infty, n)$. Therefore, the actions *send*(multicast($m$, $t$), $i$, $j$) remain enabled until they occur. And by definition of $N$, the corresponding *rcv*(multicast($m$, $t$), $i$, $j$) actions must eventually occur.

Once a *rcv*(multicast($m$, $t$), $i$, $j$) occurs, the only way for the *multicast-rcv*$_j(m)$ to be prevented is for *promises-to*$_j[k]$ to be less than $t$, for some $k \in \mathscr{I}$. Note that any new promises granted by $p_j$ must be greater than $t$ until *multicast-rcv*$_j(m)$ occurs, since $t \leq \max($*pending*$)$. Therefore, by Theorem 16 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-to*$_j[k] \geq t$, $\forall k \in \mathscr{I}$, so eventually *multicast-rcv*$_j(m)$ occurs. □

**Theorem 18.** *Module $P$ solves schedule module $M$.*

*Proof.* Follows immediately from Theorems 2, 7, 10, 11, 16, and 17 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 $P$ and any two automata that solve schedule modules $U$ and $N$. Let $\alpha$ be an execution of system $A$. We say that $\alpha$ is an *undeviating execution for* $i$ iff every pair of actions *try*$_i(S)$ and *try*$_i(S')$ 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 to whom 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 multicast 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* $\pi$: 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 $p_j$, $j \in S$, and any multicast 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 19.** *Let $\alpha$ 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 $\pi$.*

*Proof.* By Lemma 3, we know that for each $j \in S$, at most one *send*(req-promise, $i$, $j$), one *send*(promise($t$), $j$, $i$) and one *send*(multicast($m$, $t'$), $i$, $j$) occur between $\pi$ and the completion of the multicast. Now we show that at most one *send*(adv-promise($t''$), $i$, $j$) occurs. Since the execution is undeviating, promises are requested (and received) only from processes named in $S$. Since no adv-promises are sent until promises are received from all processes named in $S$, all promises are advanced at most once, to the same logical time. □

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 $|S| = n$, the promise granted by the new process each time exceeds the old *lb-time* and is 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 multicast messges are the same as above, but the number of adv-promise messages is $O(n^2)$. In situations where this sort of behavior is expected, one might choose another strategy for advancing promises. Alternative methods of promise advancement are outlined in Sect. 6.3.

### 6.2 Time complexity

To study the time complexity of the algorithm, we need a method for associating real times with points in an execution. If $\alpha$ is an execution, we say that $rt$ is a *real time assignment for* $\alpha$ if $rt$ maps each event $\pi$ in $\alpha$ to a real time $rt(\pi, \alpha)$ such that the sequence of times (1) is nondecreasing over the entire execution and (2) increases without bound if $\alpha$ is infinite. If $\alpha$ is an execution, $rt$ is a real time assignment for $\alpha$, and $\pi'$ and $\pi$ are events in $\alpha$, we say that the *time between* $\pi'$ *and* $\pi$ is $|rt(\pi, \alpha) - rt(\pi', \alpha)|$. We define the *state of* $\alpha$ *at real time* $r$ to be the state $s$ as follows: if $r$ is less than the real time of the first event in $\alpha$, then $s$ is the initial state. If $r$ is greater than the time of the last event in $\alpha$, then $s$ is the last state of $\alpha$. Otherwise, $s$ is the state occurring between the two events $\pi'$ and $\pi$ in $\alpha$ such that $rt(\pi', \alpha) < r < rt(\pi, \alpha)$. A more general approach for adding real time to the I/O automaton model is presented in [27], but the above definitions will be sufficient here.

In order to derive meaningful time bounds for the algorithm, we need to make stronger assumptions about message delivery than the eventuality conditions used for the liveness proofs. Therefore, we 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 $d$, so we do not impose any lower bound on the time between two successive steps of the algorithm. In fact, to simplify the analysis, we require that if an output action of $P$ is enabled in state $s$ at time $r$, then either that action occurs at time $r$, or that action becomes disabled by some other action occurring at time $r$. Informally, this says that the only delays are in the message system; all processing of a message occurs instantaneously with the receipt of that message. For example, no time elapses between receiving a request for a promise and sending out the promise. We also require that each user respond to *multicast-rcv* and *ready* actions immediately. That is, if a *multicast-rcv*$_i$ action occurs at real time $r$, then the resulting *try*$_i$ or *backout*$_i$ action occurs at realt time $r$. Similarly, if a *ready*$_i$ action occurs at real time $r$, then the resulting *multicast-send*$_i$ occurs at real time $r$. We will restrict our attention to executions of $A$ with real time assignments satisfying the above properties.

We wish to derive an upper bound on the time between making a request to perform a multicast (a *try*$_i$ action) and getting permission to perform the multicast (a *ready*$_i$ action). To accomplish this, we first compute an upper bound on the time for the process with the least *lb-time* to be able to perform a multicast once it has received all the necessary promises.

**Lemma 20.** *Let* $\alpha$ *be an undeviating execution for i with real time assignment* $rt$. *Let s be a state in* $\alpha$ *such that*

1. *for all* $j \in s.try\text{-}set_i$, $s.promises\text{-}from_i[j] < (\infty, n)$, *and*
2. *for all* $j \in \mathscr{I}$ *with* $s.region_j \in \{T, W\}$, $lb\text{-}time_i(s) \le lb\text{-}time_j(s)$.

*If r is the real time of state s, then there exists an event* $\pi = ready_i$ *in* $\alpha$ *such that* $r < rt(\pi, \alpha) < r + 3d$.

*Proof.* For all $j \in \mathscr{I}$, if $s.region_j \in \{\text{P, R, B}\}$ and $s.promises\text{-}to_i[j] < (\infty, n)$, then by time $r+d$, a $rcv(\text{multicast}(m, t), j, i)$ action occurs for some $m \in \mathscr{S}$ and $t \in \mathscr{T}$. Furthermore, for all $j \in \mathscr{I}$, if $s.region_j \in \{\text{T, W}\}$ and $s.promises\text{-}to_i[j] < (\infty, n)$, then a $rcv(\text{adv-promise}(t'), j, i)$ action with $t' > lb\text{-}time_i(s)$ occurs by time $r+3d$ (one delay for $p_j$'s promise requests, one delay for the promise messages, and one delay for the the adv-promise message). Any promise granted by $p_i$ after state $s$ must have a time greater than $lb\text{-}time_i(s)$, since no action can occur from region T or W to decrease the value of $lb\text{-}time_i$. Therefore, by time $r+3d$, it is the case that $\min(promises\text{-}to_i) > lb\text{-}time_i(s)$. So, all the multicast messages waiting in $pending_i$ are delivered by time $r+3d$. Thus, the preconditions for $ready_i$ are satisfied by time $r+3d$ and the action must occur. $\square$

Let $\alpha$ be an execution of $P$. We say that $p_i$ *depends on* $p_j$ in state $s$ of $\alpha$ iff $s.region_i \in \{\text{T, W}\}$, $s.region_j \in \{\text{T, W}\}$, and $lb\text{-}time_i(s) > s.promises\text{-}to_i[j]$. We say that $p_i$ *indirectly depends on* $p_k$ in state $s$ iff there is a sequence $p_i, p_{j_1}, p_{j_2}, \ldots, p_k$ such that $p_i$ depends on $p_{j_1}$, $p_{j_1}$ depends on $p_{j_2}$, etc. One may think of this sequence as a waiting chain, in which each process is waiting to receive a multicast message from the next process in the chain before it may proceed with its own multicast.

The following theorem says that if $z$ is the length of the longest waiting chain originating at $p_i$ in an undeviating execution and $p_i$ holds promises from all members of its *try-set*, then $p_i$ must wait at most $3d(z+1)$ time units before completing its multicast.

**Theorem 21.** *Let* $\alpha$ *be an undeviating execution for all* $i \in \mathscr{I}$. *Suppose that at real time r,* $p_i$ *is in state s such that* $s.promises\text{-}from_i[j] < (\infty, n)$ *for all* $j \in s.try\text{-}set_i$. *Let z be the greatest number of processes on which* $p_i$ *indirectly depends between state s and the next* $ready_i$. *Then a* $ready_i$ *occurs by time* $r + 3d(z+1)$.

*Proof.* At most time $2d$ is required from the time a process requests promises until those promises are received. Therefore, if a process $p_j$ depends on process $p_k$, it must be that $p_j$ receives a promise request from $p_k$ within time $2d$ of the $try_j$ event. (If the promise request arrived later, then $p_j$'s *lb-time* would already be fixed and $p_j$ would grant a promise for a greater time, contradicting the hypothesis that $p_j$ depends on $p_k$). So, extending this argument, the *lb-times* for all processes in the longest waiting chain originating at $p_i$ must be fixed by real time $r+2dz$. So, by Lemma 20, we know that if $p_l$ is the process in the waiting chain with the least *lb-time*, then a $ready_l$ action must occur by time $r+2dz+3d$, shortening the length of the waiting chain by one. Similarly, the next process in line must issue its *ready* action within $3d$ time units, and so on. Therefore, a $ready_i$ occurs by time $r+2dz+3dz = r+5dz$.

However, one can improve on this bound by noticing that by the end of the $3d$ maximum time units between the time the last process in the chain obtains all of its promises until its $ready_i$ occurs, all the remaining processes in the chain will have received any adv-promise messages due them. Therefore, each remaining process waits only for the multicast messages from the processes on which it directly depends. These messages require at most $d$ time units each, and there are $z$ of them in the chain. This gives us a time bound of $2dz+3d+dz=3d(z+1)$. □

It should not be surprising that the time complexity depends heavily upon pattern of the multicast requests, since this is what determines the dependency order. Since $z$ can be at most $n$, the delay is at most $3d(n+1)$.

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 protocol requires an $\Omega(dn)$ 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 far an execution without the undeviating property is rather complicated. Process $p_1$, say, grants promises to all the other processes. Then, processes $p_2$ through $p_n$ each change their minds $n$ times about their *try-set*s 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 before requesting the additional promises, $p_1$ first grants 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 *lb-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 $O(n^3 d)$.

One interesting question is whether a deeper understanding of the time complexity of the algorithm could be obtained by stating a measure of the concurrency inherent in the pattern of *try* actions and deriving a time complexity in terms of that measure. That is, one might measure how well the algorithm performs for a given pattern of multicast requests, and compare this to an optimal strategy for handling that particular pattern. Ideally, an algorithm would perform optimally for all possible request patterns. One complication in this sort of analysis is that the behavior of the protocol itself may influence the pattern of requests.

### 6.3 Possible optimizations

We begin with two simple optimizations. To simplify the presentation of the algorithm, we chose to deliver only one message in a $multicast\text{-}rcv_i$ action. As a minor modification, one might wish to send a sequence of messages in each action. Also for the sake of exposition, we chose to let $p_i$ send itself messages over the network. A real implementation, however, would not actually send such messages but simply do some local computation.

A more significant modification would involve not waiting for promises requested from processes not in one's try set. That is, $done_i$ would become enabled after $p_i$ no longer holds any promises, even if $p_i$ has requested a promise that has not yet been received. One way to achieve this would be for $p_i$ to send out "multicast" messages to every process in *requested*, regardless of whether the promise had actually 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 advancing promises. Alternatively, one might have a process request promise advancement from those processes blocking its computation. More specifically, the following options are possible.

1. 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 *lb-time*.
2. Advancement on demand: If a process $p_j$ is in T with $lb\text{-}time=t$, 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 $lb\text{-}time>t'$, then it will send $p_j$ a promise advancement message.

Deadlock avoidance methods similar to these are discussed in [26]. In both cases, there is a trade-off between the message and time complexities: as one becomes more aggressive about advancing promises to reduce time delays, the number of messages increases.

As a final modification, one might allow a process to make strategic promise requests from processes not in its *need-set*. In this way, if $u_i$ changes its mind about its *try-set*, $p_i$ may not need to wait for additional promises. Of course, requesting too many unneeded promises could adversely affect overall performance by needlessly blocking other processes.

## 7 Conclusion

We have defined the logically synchronous multicast problem and presented a solution that takes advantage of the concurrency inherent in the problem. The strong properties of message delivery order imposed by the problem would make a fault-tolerant solution highly attractive for many applications. However, in a completely asynchronous system with undetectable process failures, the properties of the message delivery order are strong enough to make a fault-tolerant solution impossible. The proof of this fact is a reduction to distributed consensus using techniques from [16]. Dolev, Dwork, and Stockmeyer show that if processes can broadcast messages such that message delivery at all processes is consistent with some total order on the broadcasts, then it is possible to implement a distributed consensus protocol that tolerates any number of stopping faults [8]. (Each process simply broadcasts its initial value, and the value in the first message received is used as the decision value). We know that there does not exist a protocol for distributed consensus that tolerates even one stopping fault [12]. Therefore, it is impossible to construct a fault-tolerant broadcast protocol in which message delivery at all processes is consistent with a single total ordering of the broadcasts. Since the logically synchronous multicast problem requires message delivery to be consistent with a total ordering of the multicasts (plus other conditions), it also does not admit a fault-tolerant solution. However, in spite of this impossibility result, there do exist useful applications of the logically synchronous multicast protocol we have presented. To conclude the paper, we illustrate an application of this protocol in an area where we need not be concerned with process failure. Specifically, 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 [6, 9, 11, 13, 15, 22, 20, 23, 21, 24, 29, 30]). Therefore, we have developed a simulation system based on that model to aid in the design and understanding of distributed algorithms [14]. *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 addition, the actions enabled in a given state of an automaton may, in general, depend upon all previous actions occurring at that automaton. Furthermore, the fairness condition requires that given an automaton $\mathscr{A}$ and an execution $\alpha$ of $\mathscr{A}$, if some class $C \in \mathrm{part}(\mathscr{A})$ has an action enabled in a state $s$ of $\alpha$, then either no action in $C$ is enabled in some state $s'$ occurring in $\alpha$ after $s$, or an action from $C$ occurs in $\alpha$ after state $s$.

We wish to construct a distributed system for simulating fair executions of a given automaton $\mathscr{A}$, where $\mathscr{A}$ has some finite number of components $\mathscr{A}_1, \mathscr{A}_2, \ldots, \mathscr{A}_n$. To simplify the discussion, we shall assume that each component $\mathscr{A}_i$ has exactly one class in its partition. (The generalization allowing each component to have a finite number of classes is straightforward.) To accomplish this, we simply "plug in" a particular transition relation for each user process $u_i$ in system $A$ such that all of its schedules are in $scheds(U_i)$: We assign process $u_i$ to simulate component $\mathscr{A}_i$. When $\mathscr{A}_i$ has an action $\pi$ enabled, $u_i$ may issue a $try_i(S)$ action, where $S$ is the set of automata having $\pi$ as an action.[10] Then, upon receiving a $ready_i$ input, $u_i$ issues a *multicast-send*$(\pi)$, where $\pi$ is the action associated with the previous $try_i$. We permit $u_i$ to issue a $backout_i$ only if no actions are enabled in $\mathscr{A}_i$. The *multicast-rcv*$_i(\pi')$ input actions are used to drive the simulation of $\mathscr{A}_i$. When a *multicast-rcv*$_i(\pi')$ action occurs, process $u_i$ updates its state based on action $\pi'$ occurring in $\mathscr{A}_i$.

Given the schedule module $M$ defined earlier, one can verify that this distributed simulation satisfies the definitions of the I/O automaton model. As far as each of the components of the simulation can tell, each action $\pi$ occurring in the simulation happens simultaneously at every component having $\pi$ 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 the correctness proof are therefore general results, independent of any particular system or application.

*Acknowledgements.* I would like to thank Hagit Attiya and Jennifer Welch for insightful dicussions, and Nancy Lynch, Mark Tuttle and Ellen Witte for their helpful comments on earlier drafts. I also thank the referees for their detailed suggestions.

## References

1. Ada programming language. Tech Rep ANSI/MIL-STD-1815A-1983, Department of Defense
2. Awerbuch B: Complexity of network synchronization. J ACM 32(4):804–823 (1985)
3. Back RJR, Kurki-Suonio R: Distributed cooperation with action systems. ACM Trans Program Lang Syst 10(4):513–554 (1988)
4. Bagrodia R: On the design of high performance distributed systems. Ph.D. dissertation, University of Texas, Austin, 1987
5. Birman KP, Joseph TA: Reliable communication in the presence of failures. ACM Trans Comput Syst 5(1):47–76 (1987)
6. Bloom B: Constructing two-writer atomic registers. IEEE Trans Comput (Special Issue on Parallel and Distributed Algorithms) 37(12):1506–1514 (1988). Also in 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 1987, pp 249–259
7. Buckley GN, Silberschatz A: An effective implementation for the generalized input-output construct of CSP. ACM Trans Program Lang Syst 5(2):223–235 (1983)
8. Dolev D, Dwork C, Stockmeyer L: On the minimal synchronism needed for distributed consensus. J ACM 34(1):77–97 (1987)
9. Fekete A, Lynch N: The need for headers: an impossibility result for communication over unreliable channels. In: Goos G, Hartmanis J (eds) CONCUR '90, Theories of concurrency: unification and extension. (Lect Notes Comput Sci, vol 458) Springer, Berlin Heidelberg New York 1990, pp 199–216
10. Fekete A, Lynch N, Mansour Y, Spinelli J: The data link layer:

[10] In a real implementation, one might have the system determine $S$ based on $\pi$

the impossibility of implementation in face of crashes. Tech Memo MIT/LCS/TM-355.b, MIT Laboratory for Computer Science, August 1989 (submitted for publication)
11. Fekete A, Lynch N, Shrira L: A modular proof of correctness for a network synchronizer. In: The 2nd International Workshop on Distributed Algorithms, July 1987. Amsterdam, The Netherlands
12. Fischer MJ, Lynch NA, Paterson MS: Impossibility of distributed consensus with one faulty process. J ACM 32(2):374–382 (1985)
13. Goldman K, Lynch N: Modelling shared state in a shared action model. In: Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science, June 1990
14. Goldman KJ: Distributed algorithm simulation using Input/Output automata. Tech Rep MIT/LCS/TR-490, MIT Laboratory for Computer Science, July 1990. Ph.D. Thesis
15. Goldman KJ, Lynch NA: Quorum consensus in nested transaction systems. In: Proceedings of the 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp 27–41, August 1987. A full version is available as MIT Tech Rep MIT/LCS/TR-390
16. Herlihy M: Impossibility and universality results for wait-free synchronization. In: Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Prinicples of Distributed Computing, pp 276–290, August 1988
17. Hoare CAR: Communicating sequential processes. Prentice-Hall, Englewood Cliffs, New Jersey, 1985
18. Joseph TA, Birman KP: Reliable broadcast protocols. In: Mullender (ed) An advanced course on distributed computing, chap 14. ACM Press, 1989
19. Lamport L: Time, clocks, and the ordering of events in a distributed system. Commun ACM, 27(7):558–565 (1978)
20. Lynch N, Merritt M: Introduction to the theory of nested transactions. In: International Conference on Database Theory, pp 278–305, Rome, Italy, September 1986. Also, expanded version in Tech Rep, MIT/LCS/TR-367, MIT Laboratory for Computer Science, July 1986. Revised version in Theor Comput Sci 62(1988):123–185
21. Lynch N, Merritt M, Weihl W, Fekete A: Atomic transactions. (in progress)
22. Lynch N, Goldman KJ: Distributed algorithms. Tech Rep MIT/LCS/RSS-5, MIT Laboratory for Computer Science, May 1989. MIT Research Seminar Series
23. Lynch N, Mansour Y, Fekete A: Data link layer: two impossibility results. In: Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp 149–170, August 1988
24. Lynch NA, Tuttle MR: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp 137–151, August 1987. A full version is available as MIT Tech Rep MIT/LCS/TR-387
25. Lynch NA, Tuttle MR: An introduction to input/output automata. CWI-Quarterly 2(3) (1989)
26. Misra J: Distributed discrete-event simulation. Comput Surv 1(18):39–65 (1986)
27. Modugno F, Merritt M, Tuttle MR: Time constrained automata. Unpublished manuscript, November 1988
28. Schneider FB: Synchronization in distributed programs. ACM Trans Program Lang Syst 2(4):179–195 (1982)
29. Welch J, Lamport L, Lynch N: A lattice-structured proof of a minimum spanning tree algorithm. In: Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp 28–43, August 1988
30. Welch J, Lynch NA: Synthesis of efficient drinking philosophers algorithms. Tech Rep MIT/LCS/TM-417, MIT, Laboratory for Computer Science, November 1989 (submitted for publication)