LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-588

CORRECTNESS PROOF FOR A NETWORK SYNCHRONIZER

Harish Devarajan
Alan Fekete
Nancy Lynch
Liuba Shrira

December 1993

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Correctness Proof for a Network Synchronizer*

Harish Devarajan* Alan Fekete† Nancy Lynch* Liuba Shrira*

December 8, 1993

Abstract

In this paper we offer a formal, rigorous proof of the correctness of Awerbuch's algorithm for network synchronization [1]. We specify both the algorithm and the correctness condition using the I/O automaton model. Our proof of correctness follows Awerbuch's intuitive arguments closely by exploiting the model's natural support for techniques of stepwise refinement and modularity. We demonstrate that the model is a powerful tool for reasoning about distributed algorithms.

*Laboratory for Computer Science, MIT, Cambridge, MA

†Department of Computer Science, University of Sydney, Australia

*This work was supported by ONR contract N00014-91-J-1988, by NSF grant CCR-8915206 and CCR-9225124 and by DARPA contract N00014-92-J-4033.

Contents

List of Figures

1 Introduction

We prove the correctnesss of a distributed protocol for network synchronization. Most networks do not offer reliable bounds on the time a message takes to arrive, so it is important to find algorithms that work correctly in an *asynchronous* system. It is, however, much easier to design algorithms if the network is *synchronous*. Awerbuch [1] proposed the use of a *synchronizer* that would enable one to convert any synchronous graph algorithm into an algorithm that performs correctly in an asynchronous (but failure-free) network. Such synchronizers have been used to give efficient asynchronous algorithms ([2]). We give a formal proof of correctness of this algorithm.

In [1], a synchronizer (called γ in that paper) is constructed for a network whose topology is any fixed, connected graph. The algorithm is described in terms of sub-algorithms that run on trees of a suitable spanning forest subgraph, and a distributed technique is given for finding a such a subgraph for which the resulting algorithm has low time and message complexity.

The synchronizing algorithm is derived as a composition of a simple synchronizer (called β) executing within each "cluster" (the set of nodes in each tree of the spanning forest subgraph), and another simple synchronizer (called α) that synchronizes between the clusters. While this description helps to explain the detailed algorithm, no formal proof of correctness is offered in [1]. We provide a formal account of an algorithm closely based on Awerbuch's description, and rigorously prove results about its correctness. Our exposition follows Awerbuch's informal arguments by building on claims that express formally the correctness of algorithms α and β that work at the level of the clusters and in between them (the spanning forest subgraph that defines these clusters is assumed to be given). The proof is modular and hierarchical, and uses the I/O automaton formalism to define the problem and to describe and prove the correctness of the algorithm offered as solution. The reader is referred to [3] for a treatment of I/O automata.

We define synchronizing behavior by specifying an I/O automaton that uses global information about the system to coordinate the behaviors of all the client automata. Our goal is to arrive at algorithms (automata) running on each of the nodes of the networks that, acting together, emulate the behavior of the global synchronizer. More precisely, we will produce a collection of automata which is such that from the point of view of each client, executions implemented by the collection are indistinguishable from those resulting from the presence of the global synchronizer.

First, we provide a specification for a global network synchronizer by giving a single I/O automaton that synchronizes across the network by exchanging messages directly with each node in the network. We then repeatedly refine this specification according to Awerbuch's construction. At each step, we will specify a collection of automata that will "implement" the synchronizing behavior of the global synchronizer in progressively "more distributed" settings. We will state and prove the corresponding correctness claims that justify each refinement, and finally arrive at a collection of automata that will run on individual nodes of the network - a collection that, we will prove, continues to synchronize the network.

After a brief description of the setting, notations and conventions in Section 1.1, we proceed in Section 2 to give a general specification for *client* automata, which are node automata that execute steps of a given synchronous graph algorithm that we wish to execute on the network. We then define the previously described global synchronizer *glob_synch*. Having thus laid out the problem and goal, in Section 2.2 we detail a collection of automata that "implements" *glob_synch*. One of the components of the collection is a "local" synchronizer (called *loc_synch* in this report), which

ensures that adjacent nodes are always within one synchronous-algorithm timestep of each other at all times.

We prove in Section 2.3 that, as a consequence, no client can distinguish *glob_synch* from the collection, and hence that the collection is, in a weak sense, an implementation of *glob_synch*.

A stronger (and conventional) notion automaton implementation involves behavior inclusion: We say that the composition C of a collection of automata implements an automaton A, if every behavior of C is a behavior of A.

It is this definition that will apply in subsequent sections when we consider the task of *implementing loc_synch* in a distributed setting.

In Section 3.1, we describe how *loc_synch* is implemented at the level of each of Awerbuch's clusters, and in between clusters. We prove that every external behavior of the composition of automata described is a behavior of *loc_synch*. In later sections, we provide distributed implementations and the necessary proofs of implementations of the intracluster and intercluster synchronizers that realize *loc_synch*. We dwell briefly on some highlights of the proof presented and end with a summary in Section 4.

1.1 Preliminaries

In this section, we describe the setting of the problem, and introduce some conventions and notations to be used throughout this report.

We are working with an *asynchronous*, failure-free network, whose topology is described by a fixed, connected, communication graph $G = (V, E)$, where the set of nodes V represents processors of the network and the set of links E represents the communication channels between them. Messages can be exchanged between pairs of neighboring nodes, and arrive in the order in which they are sent with finite but unbounded delays.

We assume a basic data type *message*. Ordered pairs of the form (m, q), where m is a *message* and $q \in V$ is a vertex, are said to be of type *tagged message*, and we refer to (m, q) as the tagged message *tagged* by q. We use ϵ to denote the null message.

We use "$*$" to denote a parameter we allow to take any permitted value of the right type (Restrictions on the the type and the values will be clear from the context).

With explicitly noted exceptions, Greek letters π, ρ and σ are used to denote actions of I/O automata, and β, β', γ and δ to represent sequences of those actions.

We call $neigh(p)$ the set $\{q \in V | (p, q) \in E\}$. Wherever used, i and j are always nonnegative integers.

To understand the functioning of client automata, we will need the idea of *preservation* of certain properties by clients. We provide a formal definition here, deferring motivations for use to the next section, where clients are discussed.

Definition 1.1 *Let M be an I/O automaton and $\mathcal{P}$ (the set of properties to be preserved) be a non-empty, prefix-closed set of sequences of actions from a set Φ satisfying $\Phi \cap int(M) = \phi$. M is said to* preserve *$\mathcal{P}$ if $\beta\pi|\Phi \in \mathcal{P}$ whenever $\beta|\Phi \in \mathcal{P}$, $\pi \in out(M)$, and $\beta\pi|M \in finbehs(M)$.*

Variables s and t denote states of automata. The following definition describes our notation for talking about states of compositions of automata.

Definition 1.2 *Let $A = \prod_i A_i$ be a composition of the collection of automata $\{A_i\}$. $\forall s \in states(A)$, $s[A_i]$ is the state of component automaton A_i when A is in state s.*

Throughout this report, unless otherwise specified, all arrays of automata state variables store values of type *boolean*, and all I/O automata implementations are such that all *boolean* variables are initially *false*, and once set to *true* by an action, remain *true* forever thereafter during that execution. With one exception (the client automata, Section 2.1.1), all I/O automata implemented in this report have no internal actions, are completely deterministic and are such that once any of their output actions is enabled, it is disabled only if it is performed.

In the next section we lay out the basic problem of network synchronization using the I/O automaton formalism.

2 The Network Synchronization Problem

In this subsection, we will formalize the network synchronization problem.

2.1 A specification of a globally synchronous system

With each node $p \in V$, we will associate an automaton *client*(p) that, we will assume, runs the rounds of the synchronous algorithm we wish to execute on the network.

In this section we will give an informal overview of synchronizing behavior and the workings of client automata and *glob_synch*, and then translate that description into the language of I/O automata.

The client automata are assumed to follow a protocol whereby, at the end of each round, they each output a tagged message set that contains all outgoing messages, and then wait for an input action delivering incoming messages before commencing computations for the subsequent round.

The automaton *glob_synch* ensures synchronous execution of the underlying algorithm by waiting until *all* the clients have output their outgoing tagged message sets before delivering incoming messages to *any* client.

We now model the above in the framework of the I/O automaton formalism. We begin with the specification of the client automata, and follow through with that of *glob_synch*.

2.1.1 Client automata

For concreteness, we assume that each client automaton's external actions consist of an input action and an output action. The output carries out round-specific messages to neighboring nodes, and the input action conveys that rounds' incoming messages from neighboring nodes to the client. Furthermore, we attribute a liveness property to the client automata by requiring them to always respond to incoming messages, whenever they are given the chance. This restriction allows us to work in the same network-failure free model that is the setting for Awerbuch's algorithm.

We make the assumption that clients are I/O automata and give below the specification of a client automaton at an arbitrary node $p \in V$ (*client*(p)) that captures the above requirements formally.

Signature

The client automaton at each node p has the action signature given below.

Input:
  client_input(p, M, i), M a set of *tagged messages*
Output:
  client_output(p, M, i), M a set of *tagged messages*
Internal:
  arbitrary

The notation for describing action signatures has a straightforward interpretation, and is used consistently throughout the remainder of this report. Consider, for example, the client's output action. Each client has an output action that, in some unspecified manner, contains three pieces of data in it to be conveyed to the automaton sharing this action with the client: A node identifier p that, in this case, identifies the client that performs the action, a tagged message set M of the indicated type that contains outgoing messages, and i, a positive integer (by convention), that is typically interpreted to be the round number of the synchronous algorithm during which the action occurred.

A similar interpretation applies to the client's input action. Internal actions of the client, which will include computations for the underlying synchronous algorithm, are left unspecified.

We continue with the description of the restrictions on the clients.

For each $p \in V$, associate a set of *p-well-formed* sequences defined as follows:

Definition 2.1 *A sequence (finite or infinite) is said to be* p-well-formed *if it consists of alternating occurrences of client_output*$(p, *, *)$ *and client_input*$(p, *, *)$*, starting with client_output*$(p, *, 1)$ *followed by client_input*$(p, *, 1)$*, such that with each occurrence of client_output*$(p, *, i)$*, i advances by 1, and with each occurrence of client_input*$(p, *, j)$*, j advances by 1.*

Let W_p be the set of p-well-formed sequences. We require the client at node p to *preserve* W_p and be such that no fair behavior of that automaton ends in *client_input*$(p, *, *)$.

By stipulating that clients preserve p-well-formedness, we ensure that clients do not send a message "out of turn" - the client is constrained to wait for incoming messages for a round before it can send out the outgoing messages for the next round. It is this requirement that supports our interpretation of the client as an automaton executing steps of a synchronous algorithm. Clients' input actions thus aid in the information flow of the underlying synchronous algorithm, *and* serves as a "clock pulse", telling the client to commence computations for a fresh round.

The constraint on the last action of a client's behavior guarantees that the client will not simply "crash" in the middle of a protocol - an requirement consistent with our assumption that the system is failure-free.

We now proceed to describe the automaton *glob_synch* that will model synchronous execution on the network.

2.1.2 The global synchronizer

The global synchronizer (depicted in figure 1) will work as follows.

After a client automaton finishes the computations that are to be carried out at that node for one round of the synchronous algorithm, it sends a *client_output* message containing all messages

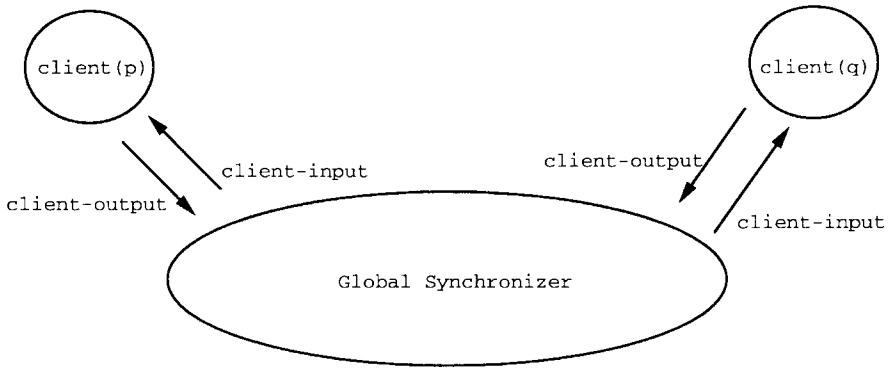

Figure 1: The Global Synchronizer and Clients

to be communicated to its neighbors in the graph to the synchronizing automaton. The automaton *glob_synch* waits until it has received similar messages from all the clients, repackages messages by destination node, sends to each client one *client_input* message containing all messages from its neighbors for that round, and waits for responses in the form of *client_output* messages of the next round.

Since each of the clients preserve the appropriate well-formedness property, the global synchronizer's functioning thus ensures synchronous execution of the underlying synchronous algorithm.

The following implementation of *glob_synch* makes it work as described.

Signature

The automaton shares each input and output action with the clients in the network, and we proceed to describe its functioning formally.

Input:
  $client\_output(p, M, i), p \in V, M$ a set of *tagged messages*
Output:
  $client\_input(p, M, i), p \in V, M$ a set of *tagged messages*
Internal:
  none

State

array $tray(p, i)$, $p \in V$, of sets of *tagged messages*, initially all empty
array $client\_output\_recd(p, i)$, $p \in V$
array $client\_input\_sent(p, i)$, $p \in V$

Transitions

$client\_output(p, M, i)$
Effect:
$s.client\_output\_recd(p, i) = true$
$\forall q\ s.tray(q, i) = s'.tray(q, i) \cup \{(m, p) | (m, q) \in M\}$

$client\_input(p, M, i)$
Precondition:
$\forall q \in V,\ s'.client\_output\_recd(q, i) = true$
$M = s'.tray(p, i)$
$s'.client\_input\_sent(p, i) = false$
Effect:
$s.client\_input\_sent(p, i) = true$

Partitions

All $client\_input(p, *, *)$ actions are in one class, for each $p \in V$.

Our interpretation of the network synchronization problem is that it is the task of implementing the *glob_synch* automaton in a distributed setting in a *fair* manner. That is, provide implementations which simulate *fair behaviors* of *glob_synch*.

In the following two sections we will provide a provably correct fair "implementation" of the composition automaton *glob_synch* and all the client automaton.

The automaton *glob_synch* works by holding back clients from executing successive steps of the algorithm until all clients have completed all message exchanges for previous steps; a "locally synchronous" implementation of *glob_synch* we are now going to consider will instead have each client wait only until all its neighbors have finished their work for the earlier rounds.

Our motivations for looking at such an implementation are twofold. Firstly, it should be clear that waiting until the *latter* condition is met is substantially easier to realize in a network. That aside, the fact that Awerbuch's synchronizer meets this weaker requirement and not that of the global synchronizer - this is intuitive - renders this implementation a natural choice owing to its potential to be refined into Awerbuch's synchronizer.

The flip side of this decision, though, is that we have to deal with a collection of automata which does not quite "implement" *glob_synch* in the usual sense (involving behavioral inclusion). What we will establish in the upcoming sections is that the collection we are going to provide (we will have called it LF by then) simulates the existence of *glob_synch* from the viewpoint of each client executing the synchronous algorithm, though each execution may itself not be globally synchronous. To be precise, we will show that for every behavior of LF interacting with client automata, there exists a behavior of the composition *glob_synch*, also interacting with clients, such that both behaviors project identically on each client - thus maintaining the semblance of global synchrony through local synchronization:

Theorem Let β be any behavior of $LF \cdot \Pi_{p \in V} Client(p)$. Then there exists $\gamma \in behs(glob\_synch \cdot \Pi_{p \in V} Client(p))$ such that for all $p \in V$, $\beta | Client(p) = \gamma | Client(p)$.

We shall forthwith proceed to the locally synchronous fair implementation of *glob_synch*.

2.2 An implementation of the global synchronizer

The automaton *glob_synch* is modeled as a collection of several automata: One front-end on each node that shares actions with the client automaton on that node, two link automata L(p $\rightharpoonup$ q) and L(q $\rightharpoonup$ p) between the front ends of every two clients that happen to be on neighboring nodes p and q, and the local synchronizer that we shall call *loc_synch*, that shares messages with all the front ends.

The front ends interface with the client automaton and receive tagged message sets from their clients that contain messages to be communicated to client automata on neighboring nodes. Each front-end sorts and distributes its own client's outgoing messages (to other front-ends, via the bridging link automata), acknowledges receipt of incoming tagged messages thus distributed and collates the incoming messages into a tagged message set. The tagged message set is then forwarded to the client if the front-end has been granted permission to do so by *loc_synch*.

When the receipt of all tagged messages sent out has been acknowledged, each front-end notifies *loc_synch* of this condition (this is the "safety" condition in Awerbuch's description). When *loc_synch* receives this signal from a front-end and all its neighbors, it authorizes that front-end to forward the tagged message set of incoming messages to its client, that is, it tells the front-end to generate the clock pulse that starts off the next round. Figure 2 depicts the various automata described and the actions they share among themselves.

We now give the formal specifications of front-end and link automata, and the local synchronizer, in that order, in each case also partitioning locally controlled actions into fairness classes.

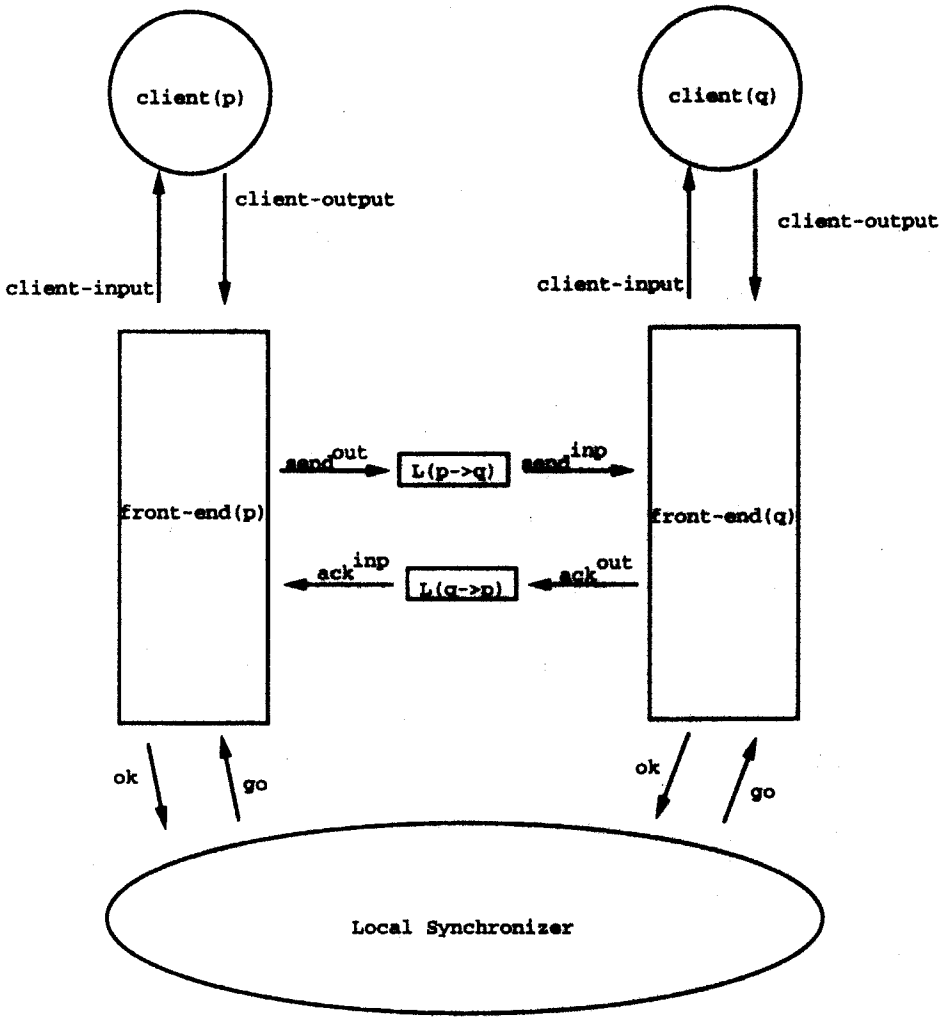

Figure 2: Automata implementing the global synchronizer

2.2.1 Front end automata

The front-end automaton ($front\_end(p)$) at any node $p \in V$ has the following formal specification. We remind the reader of our convention that, unless noted otherwise, all arrays of automata state variables store *boolean* values, and that all these values are initialized to be $false$.

Input:
- $client\_output(p, M, i), M$ a set of *tagged messages*
- $send^{inp}(q, p, \mu, i), q \in neigh(p)$, μ a set of messages
- $ack^{inp}(q, p, i), q \in neigh(p)$
- $go(p, i)$

Output:
- $client\_input(p, M, i), M$ a set of *tagged messages*
- $send^{out}(p, q, \mu, i), q \in neigh(p)$, μ a set of messages
- $ack^{out}(p, q, i), q \in neigh(p)$
- $ok(p, i)$

Internal:
- none

State

array $client\_output\_recd(i)$
array $client\_input\_sent(i)$
array $pkt\_from(q, i)$, $q \in neigh(p)$
array $pkt\_for(q, i)$, $q \in neigh(p)$
array $pkt\_sent(q, i)$, $q \in neigh(p)$
array $ack\_recd(q, i)$, $q \in neigh(p)$
array $ack\_sent(q, i)$, $q \in neigh(p)$
array $go\_recd(i)$
array $ok\_sent(i)$
array $outbox(p, i)$ of sets of *messages*, initially all empty
array $inbox(i)$, of sets of *tagged messages* , initially all empty

Transitions

$client\_output(p, M, i)$
Effect:
$s.client\_output\_recd(i) = true$
$\forall q$ such that $(m, q) \in M$, for some m
$s.outbox(q, i) = \{m | (m, q) \in M\}$
$s.pkt\_for(q, i) = true$

$send^{out}(p, q, \mu, i)$
Precondition:
$s'.pkt\_sent(q, i) = false$
$s'.pkt\_for(q, i) = true$
$\mu = s'.outbox(q, i)$
Effect:
$s'.pkt\_sent(q, i) = true$

$ack^{inp}(q, p, i)$
Effect:
$s.ack\_recd(q, i) = true$

$send^{inp}(q, p, \mu, i)$
Effect:
$s.inbox(i) = \{(m, q) | m \in \mu\} \cup s'.inbox(i)$
$s.pkt\_from(q, i) = true$

$ack^{out}(p, q, i)$
Precondition:
$s'.pkt\_from(q, i) = true$
$s'.ack\_sent(q, i) = false$
Effect:
$s.ack\_sent(q, i) = true$

$ok(p, i)$
Precondition:
$s'.client\_output\_recd(i) = true$
$\forall q \in neighbors(p)$,
if $s'.pkt\_for(q, i) = true$
then $s'.ack\_recd(q, i) = true$;
$s'.ok\_sent(i) = false$
Effect:
$s.ok\_sent(i) = true$

$go(p, i)$
Effect:
$s.go\_recd(i) = true$

$client\_input(p, M, i)$
Precondition:
$s'.go\_recd(i) = true$
$M = s'.inbox(i)$
$s'.client\_input\_sent(i) = false$
Effect:
$s.client\_input\_sent(i) = true$

Partitions

All $client\_input(p, *, *)$ actions in one class,
all $send^{out}(p, q, *, *)$ actions in one class, for each $q \in neigh(p)$,
all $ack^{out}(p, q, *, *)$ actions in one class, for each $q \in neigh(p)$,
all $ok(p, *)$ actions in one class.

2.2.2 Link automata

Front-end automata exchange messages through link automata that model communication channels between nodes in the network. We model an edge between any two nodes p and q of the network with two unidirectional link automata $L(p \rightharpoonup q)$ and $L(q \rightharpoonup p)$ which, which carry messages from one front-end to another. We specify $L(p \rightharpoonup q)$ below.

Input:
$send^{out}(p, q, \mu, i)$, μ a set of messages
$ack^{out}(p, q, i)$

Output:
$send^{inp}(p, q, \mu, i)$, μ a set of messages
$ack^{inp}(p, q, i)$

Internal:
none

State

$buffer$, a first-in first-out queue of elements of arbitrary type, initially empty

Transitions

$send^{out}(p, q, \mu, i)$
Effect:
Add $send(\mu, i)$ to $buffer$

$ack^{out}(p, q, i)$
Effect:
Add $ack(i)$ to $buffer$

$send^{inp}(p, q, \mu, i)$
Precondition:
$\text{first}(s'.buffer) = send(\mu, i)$
Effect:
$s.buffer = \text{rest}(s'.buffer)$

$ack^{inp}(p, q, i)$
Precondition:
$\text{first}(s'.buffer) = ack(i)$
Effect:
$s.buffer = \text{rest}(s'.buffer)$

Partitions

All $send^{out}(p, q, *, *)$ actions in one class,
all $ack^{out}(p, q, *)$ actions in one class.

2.2.3 The local synchronizer

As mentioned earlier, *loc_synch* ensures that a client's neighbors have all executed a round of the synchronous algorithm before allowing it to begin the next round.

Input:
$ok(p, i), p \in V$

Output:
$go(p, i), p \in V$
Internal:
none

State

array $go\_sent(p, i),\ p \in V$
array $ok\_recd(p, i),\ p \in V$

Transitions

$ok(p,i)$
Effect:
$s.ok\_recd(p,i) = true$

$go(p,i)$
Precondition:
$\forall q$ such that $q \in neigh(p) \cup \{p\}$, $s'.ok\_recd(q,i) = true$
$s'.go\_sent(p,i) = false$
Effect:
$s.go\_sent(p,i) = true$

Partitions

All $go(p,*)$ actions in one class, for each $p \in V$.

In the upcoming section, we will show how every locally synchronous execution of a synchronous algorithm corresponds to a globally synchronous execution with respect to any client.

2.3 Proof of implementation

Let LF be the automaton formed by the composition of the Local synchronizer with all the $Front\_end(p)$ automata and the links in between them, and $LC = LF \cdot \Pi_{p \in V} Client(p)$, be the "Local synchronizer and Client automata" composition. As mentioned earlier, automaton *glob_synch* differs from its "implementation" LF in that it begins a new round of communication with its clients only after they have *all* replied to the earlier set of messages. In the case of LF, on the other hand, *loc_synch* allows a client to proceed to a subsequent round as soon as the client itself and all its *neighbors* have responded to the earlier rounds' messages. In this section we will argue that this divergence in functionality is acceptable by proving that LF "implements" *glob_synch* in a precisely defined, but unconventional sense. Put in informal terms, what we will show is that as far as each client can tell, it will be running in a synchronous system.

Let GC ("Global synchronizer and Client composition") be the composition of *glob_synch* with all the client automaton. Our task will be to show that given any behavior β of LC, there exists a behavior γ of GC such that β and γ "look alike" to each of the clients. That is:

Theorem Given a behavior β of LC, there exists $\gamma \in behs(GC)$ such that for all $p \in V$, $\beta|Client(p) = \gamma|Client(p)$.

To this end, we will reorder the sequence of actions in β to get β' such that β' under an appropriate projection gives γ, a behavior of GC with the required property.

To construct the required permutation β' of β, we will define a relation that captures a dependency notion among the actions in β, and will prove that any permutation of these actions that is consistent with the relation is also a behavior of LC. We will then prove that it is possible to transform any β into a sequence of actions β', while preserving the constraints imposed by the

relation, so that β' mirrors the way GC interacts with its clients (with outgoing messages to clients being sent only after *each* client has responded to *all* previous communications). From the fact that β' is also a behavior of LC, we will be able to show that its projection γ on $acts(GC)$ yields a behavior of the latter automaton, and that β and γ project identically on each of the clients.

After proving that LC thus "implements" GC, we will justify why executions of GC thus generated are fair to its component automata, thus showing that LC results in fair implementations of GC.

2.3.1 Basic properties of the locally synchronous system

Before we state the dependency relation, we will prove some results that will allow us to reason about LF's working.

Each client's proper behavior is predicated upon its receiving appropriate inputs in response to its actions. We proceed to show that LF preserves well-formedness so that we are guaranteed that every client's behavior is also a well-formed sequence of actions.

Theorem 2.1 *For all $p \in V$, LF preserves p-well-formedness. That is, letting $\Phi_p = ext(client(p))$, for all $p \in V$, if $\beta|\Phi_p \in W_p$, and $\pi \in out(LF)$ is such that $\beta\pi|ext(LF) \in finbehs(LF)$, then $\beta\pi|\Phi_p \in W_p$.*

Proof: For all p, if $\pi \notin \Phi_p$, or $\beta|\Phi_p$ is the empty sequence of actions, the theorem holds trivially. If not, $\pi = client\_input(p, M, i)$ for some M and i. Since $\Phi_p \subseteq acts(LF)$ and $\beta\pi|ext(LF)$ is finite, $\beta|\Phi_p$ is finite too, and has a last action.
Let ρ be that last action in $\beta|\Phi_p$. We need to show that $\rho = client\_output(p, M', i)$ for some M'.

As mentioned in the section on Preliminaries, all the component automata of LF are *deterministic* and none of the automata whose composition is LF has internal actions. The projection $\beta|ext(LF) \in finbehs(LF)$ thus determines a unique execution of LF, and hence a unique state (say s') that LF will reach after that execution. Since $\beta\pi|acts(LF) \in behs(LF)$, we have, from the code implementing the *client_input* action in $front\_end(p)$, $s'[front\_end(p)].go\_recd(i) = true$. The state variable $go\_recd(i)$ is set to *true* iff $go(p, i)$ is received from *loc_synch*, and hence we examine the preconditions governing the execution of that action, in *loc_synch*.

One precondition ensures that $s'[loc\_synch].ok\_recd(p, i) = true$. (We exploit our implementational convention that boolean values, once set to *true* in an execution, remain so for the remainder of that execution). This, in turn, implies that in $front\_end(p)$, $s'[front\_end(p)].ok\_sent(i) = true$, which is predicated upon $s'[front\_end(p)].client\_output\_recd(i)$ being *true*. The action $client\_output(p, *, i)$ therefore occurs in β.

Action $client\_input(p, *, i)$'s preconditions imply that it has not occurred in β if $\beta\pi|acts(LF)$ occurs in $finbehs(LF)$. We can now infer that $\rho = client\_output(p, *, i)$. For, if not, let ρ' be the action immediately following $client\_output(p, *, i)$ in $\beta|\Phi_p$. Since $\beta|\Phi_p \in W_p$, ρ' is of the form $client\_input(p, *, i)$, which is a contradiction. ■

Another useful property of LC is that actions are never repeated in its behaviors. We prove that now.

Theorem 2.2 *In any $\beta \in behs(LC)$, for any p, q and i, each of $ack^{inp}(q, p, i), ack^{out}(p, q, i), ok(p, i)$ and $go(p, i)$ occurs at most once in β. Furthermore, for any p, q and i, there can be at most one*

action of each of the following forms in β:
$client\_output(p, *, i), send^{out}(p, q, *, i), send^{inp}(p, q, *, i), client\_input(p, *, i)$.

Proof:

Each $client\_output(p, *, i)$, for any p and i, is the output action of a client that, by definition, preserves well-formedness. It is thus guaranteed that i (which we interpret to be the round number) takes on increasing values with successive occurrences of the action, and hence that $client\_output(p, *, i)$ occurs at most once.

Every other action in β is a locally controlled action of one of the component automata of LC (since no component automaton uses internal actions). The "code" implementing each ensures non-repetition, as can be verified by inspection. ■

We will now describe the dependency relation. With each schedule of LC there is a relation $\rightarrow_\beta$ that captures the ordering that LC imposes upon the actions in β. The key property of this relation is that for any schedule β of the implementation system, and any permutation β' of β that preserves the partial order of events given by $\rightarrow_\beta$, we have that β' is also a schedule of the implementation system. In other words, $\rightarrow_\beta$ captures enough about the dependencies in the schedule to ensure that any reordering that is consistent with these dependencies is still a valid schedule of the system.

We give a formal definition of the relation $\rightarrow_\beta$.

Definition 2.2 *Given a schedule β of LC and two actions π and ρ that occur in β, the pair (π,ρ) belongs to the relation $\rightarrow_\beta$ (we write $\pi \rightarrow_\beta \rho$) if π and ρ have the form of one of the following pairs, for any $p \in V$, for any $q \in neigh(p)$, for any $q' \in neigh(p) \cup \{p\}$, for any sets of tagged messages M and M', for any set of messages μ, and for any $i \geq 0$:*

$$
\begin{gathered}
client\_input(p, M', i) \text{ and } client\_output(p, M, i+1), \\
client\_output(p, M, i) \text{ and } send^{out}(p, q, \mu, i), \\
send^{out}(p, q, \mu, i) \text{ and } send^{inp}(p, q, \mu, i), \\
send^{inp}(p, q, \mu, i) \text{ and } ack^{out}(q, p, i), \\
ack^{out}(q, p, i) \text{ and } ack^{inp}(q, p, i), \\
ack^{inp}(q, p, i) \text{ and } ok(p, i), \\
client\_output(p, M, i) \text{ and } ok(p, i), \\
ok(q', i) \text{ and } go(p, i) \text{ or} \\
go(p, i) \text{ and } client\_input(p, M, i).
\end{gathered}
$$

We write $\rho_1 \rightarrow_\beta \rho_2 \rightarrow_\beta \cdots \rightarrow_\beta \rho_r$ to mean $\rho_i \rightarrow_\beta \rho_{i+1}$ for all i, $0 < i < r$

Definition 2.3 *Given any behavior β of LC, we write $\pi \rightsquigarrow_\beta \rho$ to imply that actions π and ρ occur in that order in β, though not necessarily consecutively.*

Remark: Note that since Theorem 2.2 ensures that there are no repeated occurrences of any action, for any distinct actiona π and ρ that occur in β, exactly one of $\pi \rightsquigarrow_\beta \rho$ and $\rho \rightsquigarrow_\beta \pi$ will be true.

In lemmas to follow, we will use the definitions given above and associate the dependency relation with the relative order of occurrence of actions within a schedule. We will first show that any for two actions π and ρ of a schedule β of LF such that $\pi \rightarrow_\beta \rho$, it will also be the case that $\pi \rightsquigarrow_\beta \rho$.

Definition 2.4 *A permutation β' of $\beta \in behs(LC)$ is said to be* **consistent** *with the relation $\rightarrow_\beta$ ($\rightarrow_\beta$-consistent) if for any distinct actions π and ρ in β, $\pi \rightsquigarrow_{\beta'} \rho$ whenever $\pi \rightarrow_\beta \rho$.*

We will then use these lemmas to show that given any permutation of a schedule of the implementation automaton continues to remain a schedule if the relative ordering of actions is consistent with the dependency relation.

Lemma 2.3 *Let $\beta \in behs(LC)$. For any p, q, μ, M and i of the right type,*

1. *If $client\_output(p, M, i+1)$ occurs in β then there exists a set M' of tagged messages such that $client\_input(p, M', i) \rightsquigarrow_\beta client\_output(p, M, i+1)$.*
2. *If $send^{out}(p, q, \mu, i)$ occurs in β then there exists a set N of tagged messages such that $\{(m, q) | m \in \mu\} \subseteq N$ AND $client\_output(p, N, i) \rightsquigarrow_\beta send^{out}(p, q, \mu, i)$.*
3. *If $send^{inp}(p, q, \mu, i)$ occurs in β then $send^{out}(p, q, \mu, i) \rightsquigarrow_\beta send^{inp}(p, q, \mu, i)$.*
4. *If $ack^{out}(p, q, i)$ occurs in β then there exists a set ν of messages such that $send^{inp}(q, p, \nu, i) \rightsquigarrow_\beta ack^{out}(p, q, i)$.*
5. *If $ack^{inp}(p, q, i)$ occurs in β then $ack^{out}(p, q, i) \rightsquigarrow_\beta ack^{inp}(p, q, i)$.*
6. *If $ok(q, i)$ occurs in β then, for all $p \in neigh(q)$ for which there exists a set ν of messages such that $send^{out}(q, p, \nu, i)$ occurs in β, it will be the case that $ack^{inp}(p, q, i) \rightsquigarrow_\beta ok(q, i)$. Also, there exists a set N of tagged messages such that $client\_output(p, N, i) \rightsquigarrow_\beta ok(q, i)$.*
7. *If $go(p, i)$ occurs in β then $\forall q$ such that $q \in neigh(p) \cup \{p\}$, $ok(q, i) \rightsquigarrow_\beta go(p, i)$.*
8. *If $client\_input(p, M, i)$ occurs in β then $go(p, i) \rightsquigarrow_\beta client\_input(p, M, i)$.*

Proof:

1. This follows directly from the fact that LF and all the clients *preserve* W_p, and hence $\forall p,\ \beta | client(p) \in W_p$.
2. Verified by inspecting the code implementing actions $client\_output$ and $send^{out}$ in $front\_end(p)$.
3. $send^{inp}(p, q, \mu, i)$ is an output action of $L(p \rightharpoonup q)$. In order for this action to be enabled, $send^{out}(p, q, \mu, i)$ must have occurred at an earlier point in β.

The other cases are verified similarly. ■

The main goal of this subsection is to show that any permutation of $\beta \in behs(LC)$ that is consistent with $\rightarrow_\beta$ is also a behavior of LC. To this end, we will need the lemmas given below.

Definition 2.5 *The transitive closure of the $\rightarrow_\beta$ relation (written $\mapsto_\beta$) is defined the usual way.*

Thus $\pi \mapsto_\beta \sigma$ implies $\exists \rho_i\ i = 1, \cdots, r$, $r > 1$, such that $\rho_1 = \pi$, $\rho_r = \sigma$ and $\rho_1 \rightarrow_\beta \rho_2 \rightarrow_\beta \cdots \rightarrow_\beta \rho_r$.

Consistency of a schedule with respect to $\mapsto_\beta$ is defined in the way $\rightarrow_\beta$-consistency is, and we state the equivalence of the two notions in this lemma:

Lemma 2.4 *Every $\rightarrow_\beta$-consistent permutation of $\beta \in behs(LC)$ is $\mapsto_\beta$-consistent.*

Proof: By induction. ■

Lemma 2.5 *Every behavior β of LC is $\rightarrow_\beta$-consistent.*

Proof: Follows from Lemma 2.3.1-8. For example, let us verify that $\forall p \in V$, $\forall q \in neigh(p)$, if $ack^{inp}(q,p,i) \rightarrow_\beta ok(p,i)$, then in $\beta, ack^{inp}(q,p,i) \rightsquigarrow_\beta ok(p,i)$. Since $ack^{inp}(q,p,i) \rightarrow_\beta ok(p,i)$, $ack^{inp}(q,p,i)$ occurs in β. By successively applying Lemma 2.3.5 and Lemma 2.3.4, we get $\exists \nu$ such that $send^{inp}(p,q,\nu,i)$ occurs in β. Then by Lemma 2.3.3, $send^{out}(p,q,\nu,i)$ occurs in β. Thus, by Lemma 2.3.6, we get $ack^{inp}(q,p,i) \rightsquigarrow_\beta ok(p,i)$. Other cases follows similarly. ■

Corollary 2.6 *Every behavior β of LC is $\mapsto_\beta$-consistent.*

We will use these results to to prove the main theorem of this subsection:

Theorem 2.7 *Given $\beta \in behs(LC)$, let β' be any permutation of β consistent with $\rightarrow_\beta$. Then $\beta' \in behs(LC)$.*

Proof: To prove that $\beta' \in behs(LC)$, it suffices to show that when β' is projected on each component automaton of LC, the resulting sequences are behaviors of executions of the respective automaton [3]. We will provide the proof in the case when the component automaton is a front-end, and omit the easier proofs that establish the result for the other component automata of LC.

Our proof will be based, in part, on the following fact:

Proposition Let β be any schedule of LC. If $send^{inp}(q,p,\mu,i)$ and $client\_input(p,M,i)$ occur in β, then

$$send^{inp}(q,p,\mu,i) \mapsto_\beta client\_input(p,M,i).$$

Proof: We successively apply sub-lemmas 2.3.8 and 2.3.7, in that order, instantiating quantifiers as the notation suggests. From sub-lemma 2.3.3, $send^{out}(q,p,\mu,i)$ occurs in β and hence from 2.3.6, $ack^{inp}(p,q,i)$ occurs in β. Applying 2.3.5 and 2.3.4, we find that there exists an ν such that actions $send^{inp}(q,p,\nu,i)$, $ack^{out}(p,q,i)$, $ack^{inp}(p,q,i)$, $ok(q,i)$, $go(p,i)$, and $client\_input(p,M,i)$ all occur in β. By Theorem 2.2 $\nu = \mu$. Since each successive pair of actions in the above sequence is related by $\rightarrow_\beta$, $send^{inp}(q,p,\mu,i) \mapsto_\beta client\_input(p,M,i)$. ■

Lemma 2.8 $\forall p \in V$, $\beta'|front\_end(p) \in behs(front\_end(p))$.

Proof of lemma: The proof is by induction on the length of β'. In the case when $|\beta'| = 0$, the theorem holds trivially. Otherwise let δ be a prefix of β' such that $\delta|front\_end(p) \in behs(front\_end(p))$. Consider $\gamma = \delta\pi$, where $\pi \in acts(front\_end(p))$ is the next action in β' (We need to look only at such actions).

If π is an input action, $\gamma \in behs(front\_end(p))$, as inputs are always enabled. If not, π is one of $client\_input(p,M,i)$, $send^{out}(p,q,\mu,i)$, $ack^{out}(p,q,i)$ or $ok(p,i)$, for some q, μ, M and i. We need to show that in each case, that particular output action must be enabled. A proof for the case when $\pi = client\_input(p,M,i)$ follows. If $\pi = client\_input(p,M,i)$, by Lemma 2.3.8, $go(p,i)$

occurs before π in β. Since β' is consistent with $\rightarrow_\beta$, the same order of occurrence is maintained in γ. By Theorem 2.2 we know that no action of the form $client\_input(p, *, i)$ has occurred at an earlier point in γ. We must now show that M will have the correct value. In other words, we need this claim:

Claim $\forall q, \mu$ such that $send^{inp}(q, p, \mu, i)$ occurs in β,

$$send^{inp}(q, p, \mu, i) \rightsquigarrow_\beta client\_input(p, M, i) \text{ iff } send^{inp}(q, p, \mu, i) \rightsquigarrow_\gamma client\_input(p, M, i).$$

Proof of claim:

($\longrightarrow$)

By the Proposition and Lemma 2.4.

($\longleftarrow$)

By Corollary 2.6 and the Proposition. ■

Hence the preconditions for the $client\_input(p, M, i)$ output action of automaton $front\_end(p)$ have been satisfied, and the action is currently enabled, i.e. $\gamma\pi \in behs(front\_end(p))$.

When π is $send^{out}(p, q, \mu, i)$ for some μ, or $ack^{out}(p, q, i)$ or $ok(p, i)$, the fact that π is enabled follows immediately from the appropriate case of Lemma 2.3. ■

Similar proofs for each of the other component automata in LC show that β' projects correctly on each. Hence $\beta' \in behs(LC)$. ■

2.3.2 Reordering yields synchronized behaviors

We will now specify a way of reordering the actions of any behavior β of LC while maintaining $\rightarrow_\beta$ such that the resulting $\beta' \in behs(LC)$ will "look like" a behavior of GC. In particular, the reordered schedule will correspond to executions of LF where, at every round, front-ends send out inputs to clients only after they have confirmed that all the tagged messages in every client's output have been delivered to their destinations.

We shall call such behaviors "synchronized", and show how to construct the synchronized counterpart of any behavior of LC. In the next subsection, we will prove that upon projection onto GC, any synchronized behavior of LC will result in a behavior of GC. We will then show that given any behavior β of LC, no client will be able to distinguish β from the behavior of GC obtained by projecting onto GC the synchronized counterpart of β.

Definition 2.6 *A behavior $\beta \in behs(LC)$ is said to be a* synchronized *behavior if for all i, and for all p, q such that actions $ok(p, i)$ and $go(q, i)$ are in β, $ok(p, i) \rightsquigarrow_\beta go(q, i)$.*

Definition 2.7 *Given $\beta \in behs(LC)$, let*

- $actions(\beta, i) = \{\pi$ *occurs in* $\beta | \pi$ *is tagged with round number* $i\}$.
- $client\_inputs(\beta, i) = \{\pi | \exists M, p$ *such that* $\pi = client\_input(p, M, i)$ *occurs in* $\beta\}$.
- $oks(\beta, i) = \{\pi |$ *for some* p, $\pi = ok(p, i)\}$,
- $gos(\beta, i) = \{\pi |$ *for some* p, $\pi = go(p, i)\}$.
- $other\_acts(\beta, i) = \{\pi | \pi$ *occurs in* β *and is not in* $client\_inputs(\beta, i)$, $oks(\beta, i)$ *or* $gos(\beta, i)\}$

We now provide a way of extracting from a behavior β of LC, a synchronized behavior of LC.

Definition 2.8 *If β_i, $i = 1, 2, \cdots, n$ are finite sequences of actions, let $\bigodot_{i=1}^{n}\{\beta_i\}$ be the sequence concatenation $\beta_1\beta_2\cdots\beta_n$.*

Theorem 2.9 *For any behavior $\beta \in behs(LC)$, consider*

$$\beta' = \bigodot_{i=1}^{\infty}\{\beta|other\_acts(\beta,i)\cdot\beta|oks(\beta,i)\cdot\beta|gos(\beta,i)\cdot\beta|client\_inputs(\beta,i)\}.$$

β' is a synchronized behavior of LC.

Proof: β' is a permutation of β as every action of LC is tagged with a round number and so is an element of exactly one $actions(\beta,i)$ for some i, and $other\_acts(\beta,i)$, $oks(\beta,i)$, $gos(\beta,i)$ and $client\_inputs(\beta,i)$ partition $actions(\beta,i)$ for each i.

We need to show that β' is consistent with the $\rightarrow_\beta$ relation. First, we observe that by Lemma 2.5, β itself is consistent with $\rightarrow_\beta$ relation.

Consider relations involving actions with the same round number, say i. It can be checked that the above reordering violates none of the $\rightarrow_\beta$-consistency requirements, as none of the relations involving actions *ok*, *go* and *client_input* are violated.

Turning to orderings involving actions with different round numbers, all we have to show is that orderings of the form $client\_input(p,*,i) \rightarrow_\beta client\_output(p,*,i+1)$ are maintained. But this is true by construction.

β' is thus consistent with the $\rightarrow_\beta$. Hence, by Theorem 2.7, $\beta' \in behs(LC)$. β' is *synchronized* by construction, and this completes the proof. ■

2.3.3 Synchronized behaviors are globally synchronous

Given $\beta \in behs(LC)$, let β' be the synchronized behavior obtained after applying Theorem 2.9 to β. We will now extract from β' a behavior γ of GC which is such that with respect to each client automaton, γ and β are indistinguishable.

We will take γ to be $\beta'|acts(GC)$, and first show that

$$\gamma = \beta'|acts(GC) \in behs(GC).$$

We consider a restriction of automaton *loc_synch* that will enforce synchronized behavior. We will show that the composition of the front-ends and the restricted *loc_synch* automaton will generate all the synchronized behaviors of LC. This will be followed by a straightforward possibilities mapping proof showing that this composition implements GC. We will thus have proved that γ is a behavior of GC. We will then conclude with the proof that no client can tell γ and β apart.

Theorem 2.10 *If $\beta' \in behs(LC)$ is synchronized then $\beta'|acts(GC) \in behs(GC)$.*

Proof: Consider a restriction of *loc_synch* to *Rloc_synch*, where automaton *Rloc_synch* has the same signature and states as *loc_synch*, and executes the same transitions that *loc_synch* does on all but one of *loc_synch* actions. The automaton *Rloc_synch* will differ from *loc_synch* only in its implementation of the action $go(*,*)$. The restricted version of the *loc_synch* automaton will ensure that all front-ends are "safe" before it gives any front-end permission to send its client that rounds' input.

Rloc_synch

Input:
$ok(p,i), p \in V$

Output:
$go(p,i), p \in V$
Internal:
none

State

array $go\_sent(p,i),\ p \in V$
array $ok\_recd(p,i),\ p \in V$

Transitions

$ok(p,i)$
Effect:
$s.ok\_recd(p,i) = true$

$go(p,i)$
Precondition:
for **all** $q \in V,\ s'.ok\_recd(q,i) = true$
$s'.go\_sent(p,i) = false$
Effect:
$s.go\_sent(p,i) = true$

Partitions

All $go(p,*)$ actions in one class, for each $p \in V$.

Let

$$RLC = Rloc\_synch \cdot \prod_{p \in V} front\_end(p) \cdot client(p)$$

Then the following claim is supported by a direct proof:

Claim Every synchronized behavior of LC is a behavior of RLC.
Proof of claim: Indeed, a behavior β of LC may diverge from being a behavior of RLC only at an occurrence of $go(*,*)$ as it is this action alone that needs a different (stronger) set of preconditions satisfied in RLC. But it is this very case that is taken care of in the requirement that β be synchronized. Hence the proof follows. ■

The theorem will stand proved if we show that RLC implements GC.
Let $\Sigma = \{\pi \in acts(RLC) :\ \pi \notin acts(GC)\}$ and $\overline{RLC} = hide_{\Sigma} RLC$.

Lemma 2.11 *Consider f defined as follows:* $\forall s \in states(\overline{RLC})$,

$f(s) = \{t \in states(GC) \mid$ *for all* p *and* i,
$t.client\_output\_recd(p,i) = s[front\_end(p)].client\_output\_recd(i)$
$t.client\_input\_sent(p,i) = s[front\_end(p)].client\_input\_sent(i)$
$t.tray(p,i) = \bigcup_{r \in neigh(p)} \{(m,r) \mid s[front\_end(r)].pkt\_for(p,i) = true$ *and*
$m \in s[front\_end(r)].outbox(p,i)\}$.

Then f *is a* possibilites mapping *from* $\overline{RLC}$ *to* GC.

The above proof will need the two claims that follow. The truth of these claims are established by operational means, and are omitted here.

Claim 2.1 *For* $s \in states(\overline{RLC})$, *if* $s.[front\_end(p)].go\_recd(i) = true$ *for some* $p \in V$, *then*

1. $s.[front\_end(q)].client\_output\_recd(i) = true$ *for all* $q \in V$.
2. $s.[front\_end(p)].inbox(i) = \bigcup_{r \in neigh(p)} \{(m,r) \mid s[front\_end(r)].pkt\_for(p,i) = true$ *and*
$m \in s[front\_end(r)].outbox(p,i)\}$.

Proof of lemma: If $s \in start(\overline{RLC})$ then the existence of a start state in GC consistent with f is easily verified. Let s' be a reachable state of $\overline{RLC}$, t' any state in $f(s')$ and (s', π, s) a step of $\overline{RLC}$. We need to produce an extended step of GC (t', γ, t) such that $t \in f(s)$ and $\gamma|ext(LS) = \pi|ext(\overline{RLC})$. We handle cases differently depending on the kind of action π is.

When π is a hidden action, that is, $\pi \in \Sigma$, then γ will clearly have to be a sequence of no (zero) actions for f to be obeyed. It can be seen that no hidden action changes the values of the variables that determine the action of f. Hence we can take $t = t'$.
Otherwise, if π is an input action of $\overline{RLC}$ and GC, meaning that $\pi = client\_output(p, M, i)$ for some p, M and i, we let $\gamma = \pi$. By inspection, it can be seen that the final state t of GC will be in $f(s)$.

If not, $\pi = client\_input(p, M, i)$ for some p, M and i. Again, taking $\gamma = \pi$ and then using Claim 2.1, we can see that the preconditions of π are satisfied in GC if they are met in $\overline{RLC}$. Further, the occurrence of this action sets the *client_input_sent* variable to *true* thus ensuring the continued consistency of f.

Thus f is a possibilities mapping, and $behs(\overline{RLC}) \subseteq behs(GC)$. ■

We thus prove the main theorem of this section.

Theorem 2.12 *Let* β *be any behavior of* LC. *Then there exists* $\gamma \in behs(GC)$ *such that for all* $p \in V$, $\beta|Client(p) = \gamma|Client(p)$.

Proof: By Theorem 2.9 and Theorem 2.10 there exists β', a permutation of β such that $\gamma = \beta'|acts(GC) \in behs(GC)$. Since both β and γ are consistent with $\rightarrow_\beta$, the relative ordering of all $client\_output(p, *, *)$ and $client\_input(p, *, *)$ actions in β is maintained in γ for any p in V, and since γ is a projection of a *permutation* of β, the contents of the tagged message sets exchanged with the clients is not altered. Hence the two sequences project identically on each client. ■

We will now examine LC and sketch a proof of fairness of the implementation.

2.4 Fairness of implementation

We have shown that behaviors of LC generate behaviors of GC. Since *glob_synch* is *deterministic* in its functioning, such behaviors yield unique executions of that automaton. We will now argue why such executions obtained from *fair behaviors* of LC are *fair* executions of *glob_synch*.

Let β be any fair behavior of LC, and let γ be a behavior of GC which projects as β does on all the clients, as in Theorem 2.12. We assume also that γ is constructed as in that theorem, by considering behavior β' of RLC, and let Γ be any execution of GC that has γ as its behavior (Note that since the clients have internal actions that we have no information about, distinct executions of GC may yield indistinguishable behaviors). If Γ is not a fair execution, it can be shown that β' will not be a fair behavior of RLC. This will imply that β is not a fair behavior of LC, thus deriving a contradiction.

Combining the fairness argument with Theorem 2.12, we then have

Theorem 2.13 *Let β be any fair behavior of LC. Then there exists $\gamma \in fairbehs(GC)$ such that for all $p \in V$, $\beta|Client(p) = \gamma|Client(p)$.*

We will first show that the global synchronizer preserves the well-formedness property, and then, using this fact, prove that if the behavior γ of GC generated is unfair, then the behavior β' of RLC that γ was "extracted" from, is also unfair.

Theorem 2.14 *Automaton glob_synch preserves p-well-formedness, for all $p \in V$. That is, for each p in V, letting $\Phi_p = ext(client(p))$, if $\beta|\Phi_p \in W_p$, and $\pi \in out(glob\_synch)$ is such that $\beta\pi|ext(glob\_synch) \in finbehs(glob\_synch)$, then $\beta\pi|\Phi_p \in W_p$.*

Proof: The proof of this theorem resembles that of Theorem 2.1 strongly,

For each $p \in V$, if $\pi \notin \Phi_p$ or $\beta|\Phi_p$ is the empty sequence of actions, the theorem holds trivially. If not, $\pi = client\_input(p, M, i)$ for some M and i. Since $\Phi_p \subseteq acts(LF)$ and $\beta\pi|ext(LF)$ is finite, $\beta|\Phi_p$ is finite too, and has a last action.

Let ρ be that last action in $\beta|\Phi_p$. We need to show that $\rho = client\_output(p, M', i)$ for some M'.

Since *glob_synch* is a deterministic I/O automaton, and has no internal actions, the projection $\beta|ext(glob\_synch)$ thus determines a unique execution, and hence a unique state (say s') that *glob_synch* reaches after that execution. Since $\beta\pi|ext(glob\_synch) \in behs(glob\_synch)$, we have, from the code implementing *glob_synch*, $s'[glob\_synch].client\_output\_recd(p, i) = true$. Thus the action $client\_output(p, *, i)$ occurs in β.

Action $client\_input(p, *, i)$'s preconditions imply that it has not occurred in β if $\beta\pi|acts(LF)$ occurs in $finbehs(LF)$. We can now infer that $\rho = client\_output(p, *, i)$. For, if not, let ρ' be the action immediately following $client\_output(p, *, i)$ in $\beta|\Phi_p$. Since $\beta|\Phi_p \in W_p$, ρ' is of the form $client\_input(p, *, i)$, which is a contradiction.

■

We now proceed to the theorems that establishes the fairness properties of the implementation.

Let β' be a behavior of automaton RLC, extracted from a behavior β of LC as per Theorem 2.9. Behavior β' is a synchronized, as can be checked from the code implementing RLC. By Theorem 2.10, $\gamma = \beta'|acts(GC)$ is a behavior of GC.

Theorem 2.15 *If β' is a fair behavior of RLC, then γ is a fair behavior of GC.*

Proof:

If not, let γ be an unfair behavior of GC, and $p \in V$ be such that actions from the class of actions of the form $client\_input(p, *, *)$ are enabled but never performed in γ. Then there will be a smallest i such that $client\_input(p, *, i)$ is enabled, but never performed in Γ (an execution of GC whose behavior is γ) because once an action of that form is enabled, it is not disabled unless it is performed. It follows that $client\_input(p, *, i)$ does not occur in β'. Since the automaton $glob\_synch$ preserves well-formedness with respect to all the clients, and the clients preserve well-formedness, we hence have that $client\_output(p, *, i+1)$ will not occur in Γ, and hence not in β'. From the code implementing the automaton RLC, this in turn implies that no action of the form $client\_input(*, *, i+1)$ will occur in β' (the intuitive reason is that RLC has been built to permit front-ends to respond with a round's messages to the client only if all the clients have sent in their messages for that round), thus putting an end to the exchanges between any front-end and its client. Thus if γ is an unfair behavior of GC, then γ is finite.
We consider the case when γ is finite. If a finite behavior γ is unfair, then it follows that in the final state t of Γ, an action of the form $client\_input(p, *, i)$ is enabled, for some $p \in V$.

Let α' be an execution that has β' as its behavior. We know that in GC, $t.client\_input\_sent(p, i)$ is $false$, and hence that in $front\_end(p)$, the corresponding $client\_input\_sent$ variable is $false$ throughout that execution. In order that α' not have enabled actions in the final state s of automaton RLC, it follows that $s[front\_end(p)].go\_recd(i)$ has to be $false$. In turn, in order that loc_synch not have the go action enabled in s, $s[loc\_synch].ok\_recd(q, i) = false$ for some $q \in V$. Working backwards in like fashion, we arrive at the fact that for some $r \in neigh(q)$, $s[front\_end(q)].pkt\_for(r, i) = true$, but $s[front\_end(q)].ack\_recd(r, i) = false$. This, however, can be seen to be to imply that α' is not a fair execution of automaton RLC, thus completing the proof. ■

Thus we have proved that if γ is unfair, so is β'. We now proceed to show that if β' is unfair, β (the behavior of LC that was permuted to yield β') is not fair, thereby proving that fair behaviors of LC correspond to fair behaviors of GC, as far as each client can tell.

As before, let behavior β' of RLC correspond to an execution α' of RLC, and $\beta \in behs(LC)$ correspond to an execution α of LC. We will show that if an action is "permanently enabled" in α', then the same is true of that action in α.

Theorem 2.16 *If α is a fair execution of LC, then α' is a fair execution of RLC.*

Proof: If α' is an unfair execution of RLC, then there exists a class of actions C, and a state s in α' such that in s and all subsequent states (if any), some action from that class is always enabled.

We first consider the case when C is the class of actions of the form $client\_input(p, *, *)$ for some p in V. There will be a smallest i such that $client\_input(p, *, i)$ is enabled in state s of α' and all subsequent states, but never performed. Thus $client\_input(p, *, i)$ never occurs in α', and hence not in α as well. Since that action is enabled, it follows that $s[front\_end(p)].go\_recd(i)$ is $true$, meaning that $go(p, i)$ occurs in α' and α. Thus in α, the action $go(p, i)$ occurs, while $client\_input(p, *, i)$ never does. But this enables the latter action permanently, thus making α unfair. We have thus shown that if α' is an execution unfair to this choice of C, so is α.

Similar proofs apply when C is a class consisting of actions of any of the other possible forms: $send^{inp}$, $send^{out}$, ack^{inp}, ack^{out}, ok, go or $client\_input$. This completes the proof. ■

Thus we have shown that from the point of view of each client, fair behaviors of the implementation automata are indistinguishable from fair behaviors of the global synchronizer.

The rest of this report will deal with fashioning a provably correct fair distributed implemenation of the local synchronizer *loc_synch*.

3 Implementing a Network Synchronizer

In this section, our task will be to provide a fair, distributed implementation of the automaton *loc_synch* described in Section 2.2. Our reasons for doing so are straightforward. From Theorem 2.13 of the earlier section, we know that fair behaviors of clients and front-ends interacting with *loc_synch* "look like" fairs behaviors of GC to each client. A distributed and fair node-level implementation of automaton *loc_synch* will thus leave us with a synchronizing system whose behaviors are indistinguishable from those of GC, as far as each client is concerned.

The automaton *loc_synch* is implemented in two stages. We partition the graph into clusters of node, and assume, in the first stage, that we have some means of synchronizing all the nodes in each cluster. We then describe the process of synchronizing all the clusters. In the second stage of our implementation, we consider the problem of synchronizing the nodes inside each cluster (we provide node-level implementations) and the problem of giving node-level implementions of the intercluster synchronization process of the earlier stage. We will deal with fairness issues in each stage, thus ensuring that, at the end of the section, we have a fair and correct node-level implementation of a network synchronozer.

3.1 Cluster Level Synchronization

In this section we will describe a set of automata that will implement *loc_synch*.

The graph is divided into clusters of nodes, each with its own spanning tree (we assume that this spanning forest is constructed with some preprocessing). The synchronizing algorithm is derived as a composition of a simple synchronizer (called β in [1]) executing within each cluster, and another simple synchronizer (called α) that synchronizes between the clusters.

Each cluster implements a distributed algorithm that interfaces with the *front_end* of each *client* automaton associated with each node in the cluster. At each pulse, this algorithm, an *intracluster* synchronizer, receives and exchanges messages and detects the condition that all messages of the underlying synchronous algorithm for that pulse sent out by each node in the cluster have been received. The algorithm now passes this information on to an *intercluster* synchronizer.

When the intercluster synchronizer receives this information from all neighboring clusters and the cluster in question, it gives the intracluster synchronizer permission to allow nodes in its cluster to execute the next pulse of the synchronous algorithm. Note that this behavior closely parallels that of *loc_synch*, the automaton that enforces locally synchronous behavior between neighboring clients. A pictorial representation of the cluster level implementation follows in figure 3.

We begin by specifying the implementation automata that will carry out the above tasks, and subsequently prove the correctness of the implementation using mapping techniques. We will then briefly argue that fair behaviors of the implementation will given rise to fair behaviors of the

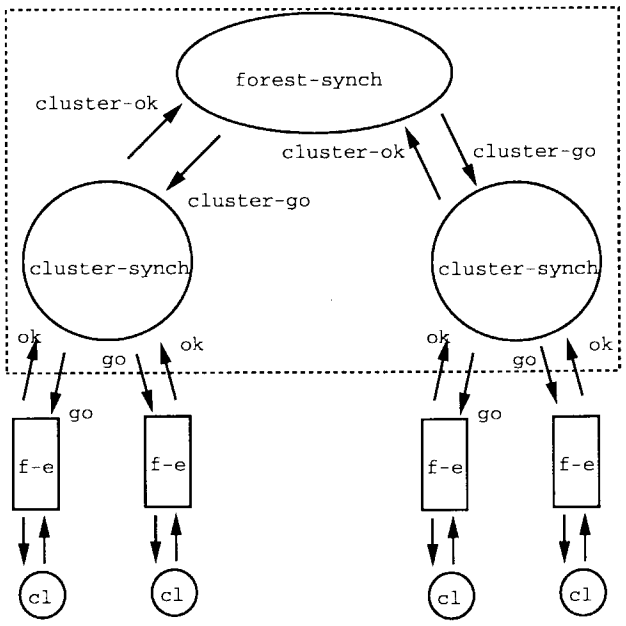

Figure 3: Automata implementing the local synchronizer

implemented system. In later sections, we will give distributed implementations of the intracluster and intercluster synchronizers described here.

We call "*forest*" the spanning forest that is assumed to be given. If C is a subtree of *forest*, *neighboring_trees*(C) is the set of trees in *forest* that edges from nodes in C are incident upon, and *nodes*(C) is the set of nodes in C. With each subtree C of *forest* we associate an automaton, an intracluster synchronizer (*cluster_synch*(C)) that does the job of synchronizing the clients on the nodes in the cluster. We give the formal specifications of an arbitrary intracluster synchronizer below.

3.1.1 Cluster level synchronizer

Automaton *cluster_synch*(C), C a subtree of *forest*:

Signature

Input:
  ok(p, i), $p \in$ *nodes*(C)
  cluster_go(C, i)
Output:
  go(p, i), $p \in$ *nodes*(C)
  cluster_ok(C, i)
Internal:

none

State

array $go\_sent(p, i)$, $p \in nodes(C)$
array $ok\_recd(p, i)$, $p \in nodes(C)$
array $cluster\_go\_recd(i)$
array $cluster\_ok\_sent(i)$

Transitions

$ok(p, i)$
Effect:
$s.ok\_recd(p, i) = true$

$cluster\_ok(C, i)$
Precondition:
$\forall p \in nodes(C), s'.ok\_recd(p, i) = true$
$s.cluster\_ok\_sent(i) = false$
Effect:
$s.cluster\_ok\_sent(i) = true$

$cluster\_go(C, i)$
Effect:
$s.cluster\_go\_recd(i) = true$

$go(p, i)$
Precondition:
$s'.cluster\_go\_recd(i) = true$
$s'.go\_sent(p, i) = false$
Effect:
$s.go\_sent(p, i) = true$

Partitions

All $go(p, *)$ actions in one class, for each $p \in nodes(C)$
all $cluster\_go(C, *)$ actions in one class.

3.1.2 A synchronizer for clusters

The intercluster synchronization part of the implementation is carried out by an automata called $forest\_synch$, as described earlier. As mentioned earlier, its functioning, captured in the formalism given below, resembles that of LS closely.

Input:
$cluster\_ok(C, i), C$ a subtree of $forest$
Output:
$cluster\_go(C, i), C$ a subtree of $forest$
Internal:
none

State

array $cluster\_ok\_recd(C, i)$ for all C a subtree of $forest$
array $cluster\_go\_sent(C, i)$, for all C a subtree of $forest$

Transitions

$cluster\_ok(C, i)$
Effect:
$s.cluster\_ok\_recd(C, i) = true$

$cluster\_go(C, i)$
Precondition:
$\forall C' \in neighboring\_trees(C) \cup \{C\}$,
$s'.cluster\_ok\_recd(C', i) = true$
$cluster\_go\_sent(C, i) = false$
Effect:
$cluster\_go\_sent(C, i) = true$

Partitions

All $cluster\_go(C, *)$ actions in one class, for each C a subtree of $forest$.

3.2 Proof of implementation

We will now prove that the compostion of $forest\_synch$ and the $cluster\_synch$ automata (call it CF) implements loc_synch, in the sense involving behavioral inclusion. The possiblities mapping proof we provide will show that every behavior of CF, when projected onto loc_synch, will yield a behavior of the latter automaton.

We first state a lemma whose truth follows directly from definitions.

Lemma 3.1 *For all p, C such that $p \in nodes(C)$,*
$\{p\} \cup neigh(p) \subseteq \{q |\ q \in nodes(C')$ *such that* $C' \in neighboring\_trees(C) \cup \{C\}\}$.

Let $\Sigma = \{\pi \in acts(CF) :\ \pi = cluster\_go(*, *)$ *or* $cluster\_ok(*, *)\}$, and $\overline{CF} = hide_{\Sigma} CF$. We will need the following lemmas for the main theorem of this section that guarantees implementation.

Lemma 3.2 *For any $p \in V$, let C be the cluster such that $p \in node(C)$.*
If $\overline{CF}$ is in state s such that $s[cluster\_synch(C)].cluster\_go\_recd(i) = true$ then
$\forall C' \in neighboring\_trees(C) \cup \{C\}, s[forest\_synch].cluster\_ok\_recd(C', i) = true$.

Lemma 3.3 *For any C a subtree of $forest$, if $\overline{CF}$ is in state s such that*
if $s[forest\_synch].cluster\_ok\_recd(C, i) = true$, then $\forall q \in nodes(C)$,
$s[cluster\_synch(C)].ok\_recd(q, i) = true$.

Both lemmas follow from operational proofs which are omitted here.

Using these lemmas, we will show a possibilities mapping between $\overline{CF}$ and *loc_synch* which will prove that every behavior of $\overline{CF}$ is a behavior of *loc_synch*. Since the behaviors of $\overline{CF}$ are plainly those of CF projected onto *loc_synch*, we will have proved the correctness of the implementation.

Theorem 3.4 *Consider f defined as follows:* $\forall s \in states(\overline{CF})$,
$f(s) = \{t \in states(loc\_synch) \mid \forall p \in V$

$$t.go\_sent(p,i) = s[cluster\_synch(C)].go\_sent(p,i) \text{ and}$$
$$t.ok\_recd(p,i) = s[cluster\_synch(C)].ok\_recd(p,i),$$

where C is the cluster such that $p \in nodes(C)$}
Then f is a possibilities mapping *from $\overline{CF}$ to loc_synch.*

Proof: If $s \in start(\overline{CF})$ then the existence of a start state in *loc_synch* consistent with f is easily verified. Let s' be a reachable state of $\overline{CF}$, t' any state in $f(s')$ and (s', π, s) a step of $\overline{CF}$. We need to produce an extended step of *loc_synch* (t', γ, t) such that $t \in f(s)$ and $\gamma|ext(LS) = \pi|ext(\overline{CF})$. We have the following cases depending on π:

- $\pi = ok(p,i)$ for $p \in V$. Then take $\gamma = \pi$. Since $ok(p,i)$ is an input action, for some t, (t', γ, t) is a step of LS. From the code describing the transitions of $LS$and *cluster_synch*(C) (as above), and given that $t' \in f(s')$ we can infer that $t \in f(s)$.

- $\pi = go(p,i)$ for $p \in V$. Again $\gamma = \pi$. If $go(p,i)$ is an enabled output action in s', it is an enabled action in *loc_synch* as well: We need to show that $t'.go\_sent(p,i) = false$ and that $\forall q$ such that $q = p$ or $q \in neigh(p)$, $t'.ok\_recd(q,i) = true$. The first condition is guaranteed by the mapping. We establish the truth of the second proposition now. Since $go(p,i)$ in enabled in $\overline{CF}$, $s'[cluster\_synch(C)].cluster\_go\_recd(i) = true$.

  From Lemmas 3.2, 3.3 and 3.1, the corresponding *ok_recd* variables in $t'[loc\_synch]$ are *true*. $go(p,i)$ is therefore enabled when *loc_synch* is in state t'. Let t be the state after $go(p,i)$ has been performed. Clearly $t \in f(s)$, and the mapping is legal.

- $\pi =$ *cluster_go*$(*,*)$ *or cluster_ok*$(*,*)$. Since these are both hidden actions, the mapping must leave t unchanged, with γ being a sequence of zero actions. This is appropriate as neither action affects variables of the form $s[*].go\_sent(*,*)$ or $s[*].ok\_recd(*,*)$, which determine the mapping.

■

After having thus shown that behaviors of CF project onto *loc_synch* as the latter automaton's behavior, we proceed now to argue that *fair* behaviors of CF will generate fair behaviors of *loc_synch* too.

3.3 Fairness of Implementation

Let β be a behavior of CF interacting with clients, their front-ends and all the link automata, and γ be such that $\gamma = \beta|loc\_synch$ is a behavior of *loc_synch*. As before, we let Γ be the unique

execution that γ is the behavior of (Γ's uniqueness is a consequence of the implementing automata being deterministic, and of their not having internal actions).

We will show that Γś being unfair will imply that β is not a fair behavior. We thus derive a contradiction to our assumption that β is a fair behavior of CF, and complete the fairness analysis.

Theorem 3.5 *If β is a fair behavior of CF interacting with clients, their front-ends and the bridging link automata, then $\gamma = \beta|loc\_synch$ is a fair behavior of loc_synch.*

Proof: We know from Theorem 3.4 that γ is a behavior of *loc_synch*. Assume that Γ the unique execution of that has γ as its behavior, is not a *fair* execution of *loc_synch*. Then there is a smallest i such that for some $p \in V$, $go(p, i)$ is enabled, but never occurs in Γ.

Since for all $j < i$, $go(q, j)$ occurs in Γ for each q in V, the action $client\_input(q, *, i-1)$ is enabled at some point in β, and will occur in β. Γ. By Theorem 2.1 actions $client\_output(q, *, i)$ actions therefore occur in β for each $q \in V$, and hence in Γ as well.

Let p be in the cluster C in *forest*. If action $go(p, i)$ does not occur in CF, it follows that for some node q in C or one of its neghboring clusters (that is, the set $\{r|\ r \in nodes(C')$ such that $C' \in neighboring\_trees(C) \cup \{C\}\})$, $ok(q, i)$ does not occur in Γ either, as otherwise β would be unfair too. But if $ok(q, i)$ is never enabled in Γ even after $client\_output(q, *, i)$ has occurred, it follows that $client(q)$'s packet meant for a neighbor is never acknowledgded. This however, can happen only if β is unfair, thus deriving a contradiction. ■

In subsequent sections we will provide distributed implementations of the *cluster_synch* and *forest_synch* automata that are formal specifications of synchronizer β (at the intracluster level) and α (at the intercluster level) respectively, as described in Awerbuch's paper. Note that we can, in principle, use the distributed versions of arbitrary synchronizers to perform the task at the two levels. All we have to ensure is that the *cluster_ok* and *cluster_go* actions shared by the two specification automata can be communicated in the network between the implementations. In this paper we restrict ourselves to implementing α and β synchronizers that Awerbuch uses, where the problem of communicating is simply solved (as will be made clear shortly) by having the automata that share these actions run on the same physical node in the network.

We close this section with a a brief overview of the remainder of the report.

The next section will detail the distributed implementation of *cluster_synch* automata, and the one after that, a mapping proof that will prove the implementation correct. This will be followed by an implementation of the automaton *forest_synch*, and the corresponding correctness proof in Sections 3.7 and 3.8. Finally, we end with a summary in Section 4.

3.4 Intracluster Synchronization

Our task in this section will be to give a distributed implementation of the intracluster synchronizer $cluster\_synch(C)$ operating in a cluster consisting of nodes of a subtree C of the given spanning forest of G.

Automaton *cluster_synch*'s function is to detect, after the commencement of each round, the condition that messages of the underlying synchronous algorithm sent out by each node in its cluster have been received. That is, detect the condition that the front ends at all the nodes in the cluster have output an *ok* action for that round.

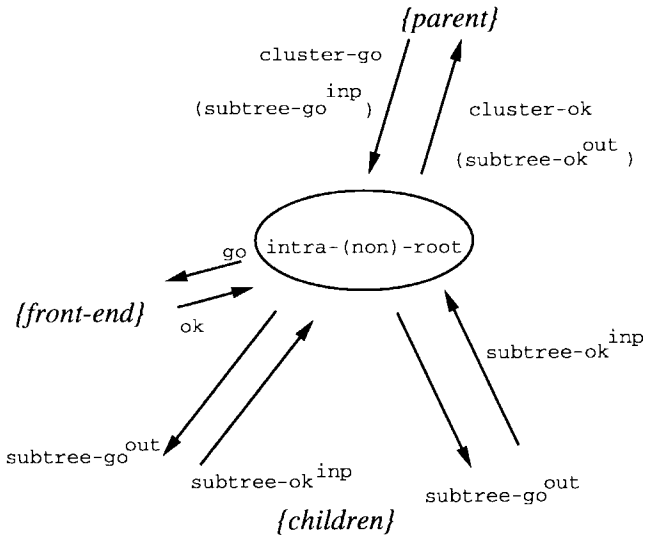

Figure 4: (Non-)Root Node Automaton of the intracluster synchronizer

At each non-root node of C, we have a component automaton *intra_non_root* of our distributed implementation sharing the *ok* with the front end at that node. Whenever an *intra_non_root* automaton learns that its front-end has sent its *ok*, and that the same is true of all the front ends at all descendent nodes, it communicates this fact to its parent in C. The process of conveying "*ok*ness" thus commences at the leaf nodes of C, and terminates at the root node of C. At the root node another component automaton (we shall refer to it as the "*intra_root*" automaton) waits until the convergecast terminates, and then outputs a *cluster_ok* action indicating that all nodes in its tree have output their *ok*s for that round.

An additional task performed by *cluster_synch* is to communicate, to all the nodes in its cluster, the receipt of the *cluster_go* action from the intercluster synchronizer. This is realized by broadcasting a message indicating receipt down the tree, from parent nodes to daughter nodes, until all the leaves have received this information.

We now give a concise description of the the compostion's functions in terms of actions exchanged between constituent components.

The *intra_root* automaton at the root node of each cluster waits until all the ok's of the nodes in the cluster have been convergecast to it, and then output a *cluster_ok*. Upon receiving a *cluster_go*, it broadcasts this information down all subtrees. The *intra_non_root* automata, one at every non-root node of a cluster, assist in the above by conveying the *ok*ness of the front end at that and at all descendent nodes up the tree, and by broadcasting the *subtree_go* message output by the root node when *intra_root* receives a *cluster_go* (see fig 4.)

If C is a subtree and $p \in nodes(C)$, let $children(p, C)$ denote p's children in C, and $parent(p, C)$ be p's parent in C (if p is not the root of the subtree). Also, let $root(C)$ denote the root node of C.

We provide formal specifications of the described automata.

3.4.1 Root node automata of the intracluster synchroniser

Automaton $Intra\_root(p, C)$, $p = root(C)$

Signature

Input:
- $ok(p, i)$
- $subtree\_ok^{inp}(q, p, i)$, $q \in children(p, C)$
- $cluster\_go(C, i)$

Output:
- $go(p, i)$
- $subtree\_go^{out}(p, q, i)$, $q \in children(p, C)$
- $cluster\_ok(C, i)$

Internal:
- none

State

array $ok\_recd(i)$
array $go\_sent(i)$
array $subtree\_ok\_recd(q, i)$, $q \in children(p, C)$
array $subtree\_go\_sent(q, i)$, $q \in children(p, C)$
array $cluster\_ok\_sent(i)$
array $cluster\_go\_recd(i)$

Transitions

$ok(p, i)$
Effect:
$s.ok\_recd(i) = true$

$subtree\_ok^{inp}(q, p, i)$
Effect:
$s.subtree\_ok\_recd(q, i) = true$

$cluster\_ok(C, i)$
Precondition:
$\forall q \in children(p, C)$, $s'.subtree\_ok\_recd(q, i) = true$
$s'.ok\_recd(i) = true$
$s'.cluster\_ok\_sent(i) = false$
Effect:
$s.cluster\_ok\_sent(i) = true$

$cluster\_go(C, i)$
Effect:
$s.cluster\_go\_recd(i) = true$

$go(p, i)$
Precondition:
$s'.cluster\_go\_recd(i) = true$
$s'.go\_sent(i) = false$
Effect:
$s.go\_sent(i) = true$

$subtree\_go^{out}(p, q, i)$
Precondition:
$s'.cluster\_go\_recd(i) = true$
$s'.subtree\_go\_sent(q, i) = false$
Effect:
$s.subtree\_go\_sent(q, i) = true$

Partitions

All $go(p, *)$ actions in one class,
all $subtree\_go^{out}(p, q, *)$ actions in one class, for each $q \in children(p, C)$,
all $cluster\_ok(C, *)$ in one class.

3.4.2 Non-root node automata in the intracluster synchronizer

Automaton $Intra\_non\_root(p, C)$, $p \in nodes(C)$, $p \neq root(C)$

Note that the only way *intra_root* and *intra_non_root* differ is that the former has *cluster_go* as an input, while the latter has *subtree_go* instead, and that the root automaton has *cluster_ok*

as an output unlike *intra_non_root*, which has *subtree_ok* instead. Functionally however, the two actions within each pair are identical, differing only in their automata of origin or destination.

Input:
$ok(p, i)$
$subtree\_ok^{inp}(q, p, i)$, $q \in children(p, C)$
$subtree\_go^{inp}(r, p, i)$, $r = parent(p, C)$

Output:
$go(p, i)$
$subtree\_go^{out}(p, q, i)$, $q \in children(p, C)$
$subtree\_ok^{out}(p, r, i)$, $r = parent(p, C)$

Internal:
none

State

array $ok\_recd(i)$
array $go\_sent(i)$
array $subtree\_ok\_recd(q, i)$, $q \in children(p, C)$
array $subtree\_go\_sent(q, i)$, $q \in children(p, C)$
array $subtree\_ok\_sent(i)$
array $subtree\_go\_recd(i)$

Transitions

$ok(p, i)$
Effect:
$s.ok\_recd(i) = true$

$subtree\_ok^{inp}(q, p, i)$
Effect:
$s.subtree\_ok\_recd(q, i) = true$

$subtree\_ok^{out}(p, r, i)$
Precondition:
$\forall q \in children(p, C)$, $s'.subtree\_ok\_recd(q, i) = true$
$s'.ok\_recd(i) = true$
$s'.subtree\_ok\_sent(i) = false$
Effect:
$s.subtree\_ok\_sent(i) = true$

$subtree\_go^{inp}(r, p, i)$
Effect:
$s.subtree\_go\_recd(i) = true$

$go(p, i)$
Precondition:
$s'.subtree\_go\_recd(i) = true$
$s'.go\_sent(i) = false$
Effect:
$s.go\_sent(i) = true$

$subtree\_go^{out}(p, q, i)$
Precondition:
$s'.subtree\_go\_recd(i) = true$
$s'.subtree\_go\_sent(q, i) = false$
Effect:
$s.subtree\_go\_sent(q, i) = true$

Partitions

All $go(p, *)$ actions in one class,
all $subtree\_ok^{out}(p, r, *)$ actions in one class, for $r = parent(p)$,
all $subtree\_go^{out}(p, q, *)$ actions in one class, for each $q \in children(p, C)$.

We model the tree edges with I/O automata, and describe a simple automaton that represents an asynchronous bidirectional link between a parent and a daughter node that carries messages between their $intra(\_non)\_root$ automata.

3.4.3 Tree link automata

Automaton $Intra\_tree\_link(p \rightharpoonup q, C)$, $p = parent(q, C)$

Signature

Input:
$subtree\_go^{out}(p, q, i)$
$subtree\_ok^{out}(q, p, i)$
Output:
$subtree\_go^{inp}(p, q, i)$
$subtree\_ok^{inp}(q, p, i)$
Internal:
none

State

$buffer$ a first-in first-out queue of elements of arbitrary type, initially empty.

Transition

$subtree\_go^{out}(p, q, i)$
Effect:
Add $go(q, i)$ to $buffer$

$subtree\_ok^{out}(q, p, i)$
Effect:
Add $ok(q, i)$ to $buffer$

$subtree\_go^{inp}(p, q, i)$
Precondition:
first$(s'.buffer) = go(q, i)$
Effect:
$s.buffer =$ rest$(s'.buffer)$

$subtree\_ok^{inp}(q, p, i)$
Precondition:
first$(s'.buffer) = ok(q, i)$
Effect:
$s.buffer =$ rest$(s'.buffer)$

Partitions

All $subtree\_go^{inp}(p, q, *)$ actions in one class,
all $subtree\_ok^{inp}(q, p, i)$ actions in one class.

In the next section we justify using the described collection to implement *cluster_synch* by providing a proof of implementational correctness.

3.5 Proof of implementation

We will now show that the intra_root automaton, working with the intra_non_root automata and the links, implements $cluster\_synch(C)$. Let $TRA(C)$ be the composition of the $intra\_root(p, C)$, $p = root(C)$ automaton with all the $intra\_non\_root(*, C)$ automata and the $intra\_tree\_link(* \rightharpoonup *, C)$

automata. Further, let $intra(r, C)$ denote $intra\_root(r, C)$ or $intra\_non\_root(r, C)$ depending on whether or not $r = p = root(C)$.
Let $\overline{TRA(C)} = hide_{\Sigma} TRA(C)$, where
$\Sigma = \{\pi \in acts(TRA(C)) | \pi$ is a $subtree\_ok^{inp}$, $subtree\_ok^{out}$, $subtree\_go^{inp}$, or $subtree\_go^{out}$ action$\}$

The following theorem guarantees that every behavior of $\overline{TRA(C)}$ is a behavior of $cluster\_synch(C)$.

Theorem 3.6 *$\forall s \in states(\overline{TRA(C)})$ let f be as below.*

$f(s) = \{t \in states(cluster\_synch(C)) |$ $\forall r \in nodes(C)$
$t.ok\_recd(r,i) = s[intra(r,C)].ok\_recd(i)$
$t.go\_sent(r,i) = s[intra(r,C)].go\_sent(i)$
$t.cluster\_ok\_sent(i) = s[intra\_root(p,C)].cluster\_ok\_sent(i)$
$t.cluster\_go\_recd(i) = s[intra\_root(p,C)].cluster\_go\_recd(i)$ $\}$

Then f is a possibilities mapping *from $TRA(C)$ to $cluster\_synch(C)$.*

Proof:

If $s \in start(\overline{TRA(C)})$ the existence of the start state in $cluster\_synch(C)$ consistent with f is easily verified. Let s' be a reachable state of $\overline{TRA(C)}$, t' any state in $f(s')$ and (s', π, s) a step of $\overline{TRA(C)}$. We need to produce an extended step of $cluster\_synch(C)$ (t', γ, t) such that $t \in f(s)$ and $\gamma | ext(cluster\_synch(C)) = \pi | ext(\overline{TRA(C)})$.

- $\pi \in \Sigma$
  Taking γ to be the null sequence of actions, and noting that for all steps (s', π, s) such that $\pi \in \Sigma$ $f(s') = f(s)$, we have f being a valid mapping, as $t = t' \in f(s') = f(s)$

- $\pi \in inputs(cluster\_synch(C))$
  We take $\gamma = \pi$. Since γ is an input, it follows that (t', γ, t) is a step (t is the state $cluster\_synch(C)$ winds up in after γ). That $t \in f(s)$ can be verified by inspection in both the two cases ($\pi = ok(p, i)$ or $cluster\_go(C, i)$).

- $\pi = go(r, i)$, $r \in nodes(C)$
  As before, take $\gamma = \pi$. If $r = p$, legality of f follows directly from the code implementing this action in loc_synch and $intra\_root(r, C)$. If not, π is output by $intra\_non\_root(r, C)$, which would imply that $s'[intra\_non\_root(r, C)].subtree\_go\_recd(i) = true$ and $s'[intra\_non\_root(r, C)].go\_sent(i) = false$. From the second equality, we immediately have $t'[cluster\_synch(C)].go\_sent(p, i) = false$. All we have to show is that when $cluster\_synch(C)$ reaches state t' it has already received the $cluster\_go$ for the ith round.

  Claim 3.1 *For any $r \in nodes(C), r \neq root(C)$, let q_j, $j = 0, 1, \cdots, d$ be the nodes in C such that $q_0 = r$, $q_d = root(C)$ and $\forall j < d$, $q_{j+1} = parent(q_j, C)$.*
  If $\overline{TRA(C)}$ is in state s' such that $s'[intra\_non\_root(q_0, C)].subtree\_go\_recd(i) = true$, then

  1. $\forall j < d, s'[intra\_non\_root(q_j, C)].subtree\_go\_recd(i) = true$
  2. $s'[intra\_root(q_d, i)].cluster\_go\_recd(i) = true$.

Proof of claim:

1. By induction on j.
2. From Claim 3.1.1 $s'.[intra\_non\_root(q_{d-1}, C)].subtree\_go\_recd(i) = true$. From the implementation of the *intra_root* automaton, the claim follows.

■

Action γ will thus be enabled in t' and t such that (t', γ, t) is a step of $cluster\_synch(C)$ will be in $f(s)$.

- $\pi = cluster\_ok(C, i)$
  Taking $\gamma = \pi$. If *intra_root* receives *subtree_ok* messages from all its children, it needs to be shown that at the *intra_non_root* automata at all descendent nodes, the *ok_recd* received variable for that round is *true*. That γ will be enabled in $cluster\_synch(C)$ will follow immediately, and the proof would be done.

  Claim 3.2 $\forall s' \in states(\overline{TRA(C)})$ *if* $s'[intra\_root(p, C)].cluster\_ok\_sent(i) = true$ *then* $\forall r \in nodes(C)$, $s'[intra(r, C)].subtree\_ok\_sent(i)] = true$

  Proof of claim: For any node r in C, let $depth(r)$ denote its distance from $root(C)$, along the unique path entirely in C between r and $root(C)$. We prove the claim by induction on the depth of a node r. For a node r at depth 1, since r is a child of the root node, $s'[intra\_root(p, C)].cluster\_ok\_sent(i) = true$ implies that $s'[intra(r, C)].subtree\_ok\_sent(i) = true$.
  Assume that at any node r' at a depth $d-1$, $s'[intra(r', C)].subtree\_ok\_sent(i)] = true$. Consider a node r at a depth d, which has q at depth $d-1$ as the parent. The required condition follows directly from the implementation of the $subtree\_ok^{out}(r, q, i)$ action.
  ■

  From the above claim and the preconditions of the $subtree\_ok^{out}$ action we have, for all nodes q, $s'[intra(q, C)].ok\_recd(i) = true$, as required.

  ■

We are thus guaranteed that $behs(\overline{TRA(C)}) \subseteq behs(cluster\_synch(C))$. But behaviors of $\overline{TRA(C)}$ are clearly projections of behaviors of $TRA(C)$ onto $cluster\_synch(C)$. Hence, $TRA(C)$ implements $cluster\_synch(C)$.

3.6 Fairness of Implementation

We now argue that TRA is a fairness preserving implementation. Automaton $cluster\_synch(C)$, for some subtree C in *forest*, has $cluster\_ok(C, *)$ and $go(p, *)$ for each $p \in nodes(C)$ as fairness partitions. As in earlier fairness arguments, it can be shown that unfair behaviors of $cluster\_synch(C)$ can be generated only from unfair behaviors of $TRA(C)$.

Theorem 3.7 *Let β be a fair behavior of $TRA(C)$, for some subtree C in forest. Then $\gamma = \beta | cluster\_synch(C)$ is a fair behavior of automaton $cluster\_synch(C)$.*

Proof:

Theorem 3.6 guarantees that γ is a behavior of *cluster_synch*(C). Assume that γ is an unfair behavior of *cluster_synch*(C), and let Γ be the unique execution of that automaton which has γ as its behavior.

Consider the case when the action $go(p, i)$ is enabled in Γ but never performed, for some i and p in V. If p is the root node of the subtree C, a contradiction is immediate: The enabling of $go(p, i)$ implies that *cluster_go*(C, i) occurs in γ, and hence in β as well. In that event, $go(p, i)$ is enabled in the execution corresponding to β as well, in the automaton *intra_root*(p, C). Since β is a fair behavior, this results in that action being performed, which is a contradiction.

If p is a non-root node of C, at a depth $d > 1$ from the root node, a inductive argument yields a contradiction: If $go(p, i)$ does not occur in the fair behavior β of $TRA(C)$, it follows that it is never enabled during that execution. Automaton *intra_non_root*(p, C) therefore never receives a *subtree_go* message from the automaton *intra*$(parent(p, C), C)$ at level $d - 1$. This can happen only if *intra*$(parent(p, C), C)$ never receives a *subtree_go* from *its* parent. Thus if p is such that $go(p, i)$ is enabled but never performed in Γ, then it follows by induction that the the *intra_non_root* automaton at depth 1 on the path joining the root to p never receives a *subtree_go* message. Since this can happen only if *cluster_go*(p, i) is not performed, we have a contradiction.

We now consider the case when, for some i, the action *cluster_ok*(C, i) is enabled, but never performed in Γ. If *cluster_ok* is enabled in *cluster_synch*(C), then it follows that all the front-ends in C have sent in their *ok*s for that round to *cluster_synch*(C). Thus all these *ok*s appear in β as well.

If *cluster_ok* is not performed in the execution corresponding to β (in $TRA(C)$), it would have to have been disabled throughout that execution, for otherwise β would not be fair. This implies that one of the root node's child nodes never sends in its *subtree_ok* action for that round. By the same argument, if a node at depth d does not ever receive a *subtree_ok* from one of its children for some round, this can happen only if that child node did not receive such an action from one of *its* children, at depth $d + 1$.

This inductive argument leads to the fact that at least one leaf node *intra_non_root* automaton never sends in a *subtree_ok* action for that round. But every descending branch of the subtree ends in a leaf node whose *subtree_ok* action is enabled if all the *ok*s for that round appear in β, and, if β is fair, the automaton at this node does perform the *subtree_ok* action at some point in β. We thus derive a contradiction to our assumption that fair behaviors of $TRA(C)$ can generate unfair behaviors of *cluster_synch*(C). ■

We saw in an earlier section that *loc_synch* can be implemented as a combination of an intra-cluster synchronizer (as described in this section) and an intercluster synchronizer that synchronizes between clusters. We now direct our attention to the latter automaton, and put together a collection of automata that will implement the intercluster synchronizer *forest_synch* fairly.

3.7 Intercluster Synchronization

We will describe a collection of automata that will implement *forest_synch*. This automaton's function resembles that of *loc_synch* rather closely: It detects the condition that a cluster and all its neighboring clusters are "safe" (have sent in their *cluster_ok*s) and communicates this fact to

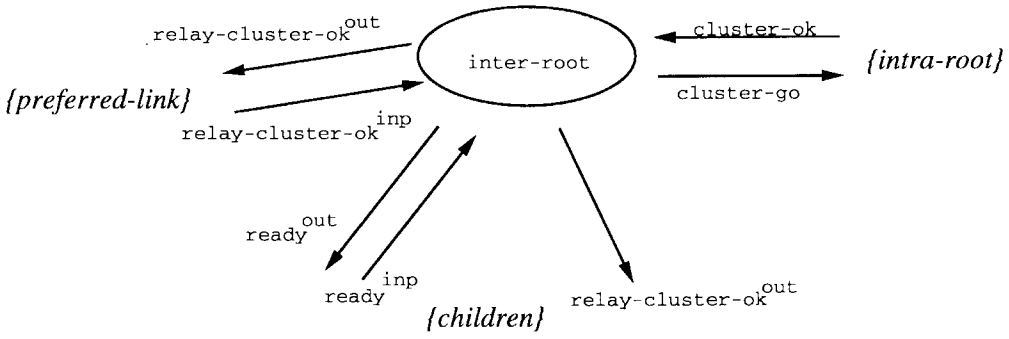

Figure 5: The root-node component automaton of the intercluster synchronizer.

the intracluster synchronizer for that cluster.

This requires each intracluster to share actions with a component of *forest_synch*'s distributed implementation. In our (and Awerbuch's) implementation, this constraint is met by having a key component automaton (that we shall call "*inter_root*") of *forest_synch*'s implementation run on each root-node of a cluster (which has the *intra_root* automaton that generates the *cluster_ok* action, and receives the *cluster_go* action).

Communication between any two neighboring clusters in the distributed implementation occurs through a predetermined set of edges that form a path between the root nodes of the two cluster. All but one of the path's edges are chosen from the spanning trees of either cluster. We call the non-tree edge, the *preferred_link* between the two clusters, and denote the set of all such cluster-bridging edges by *preferred_links*(*forest*).

We now describe the operation of the distributed implementation.

The distributed implementation of *forest_synch* consists of the following in *each* cluster: a *inter_root* automaton at the root of the cluster, and a *inter_non_root* automaton at *each* non-root node of the cluster.

The functions of the *inter_root* automaton are twofold.

- Upon being told by *cluster_synch* that the cluster is safe, it will broadcast this down to the leaves, through the subtrees, so that neighboring clusters can get this information via bridging *preferred links.*
- Messages from neighboring clusters telling the cluster in question that they are safe are convergecast to the *inter_root* automaton. After checking to see if its own cluster is safe, it will then give the *cluster_synch* automaton (also at the root node) a *cluster_go.*

Figure 5 summarizes the *inter_root* automaton's functioning pictorially.

The *inter_non_root* automaton's functioning resembles that of *inter_root* closely. Upon receiving *relay_ok_recd* passed down to it along the path joining it to *inter_root*, *inter_non_root* passes

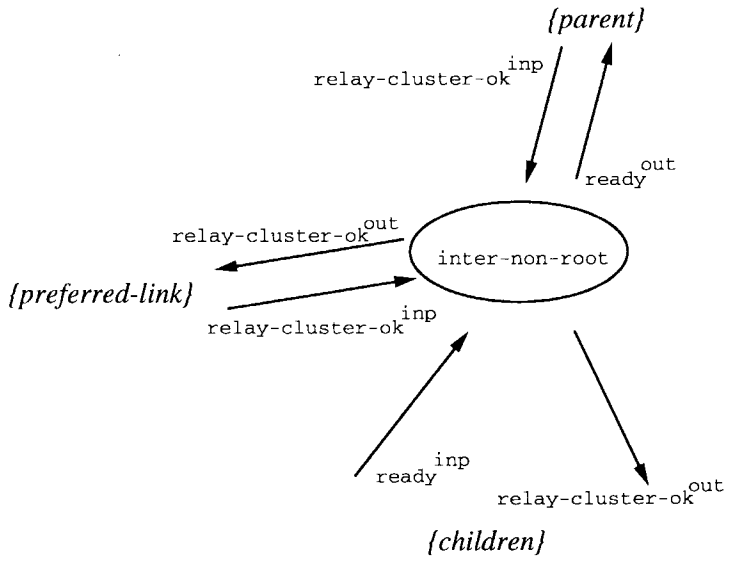

Figure 6: The non-root node component automaton of the intercluster synchronizer.

it along to its children, and any preferred links that may be incident upon it. This behaviour corresponds to that exhibited by the root automaton when it receives a *cluster_safe* from *cluster_synch*. Another important task performed by *inter_non_root* is to help convergecast *readiness* to the root. An *inter_non_root* automaton is *ready* if

- All its children are *ready* and
- if it has received *relay_cluster_ok*s along all preferred edges incident upon it.

This is merely the condition that the cluster itself, all clusters neighboring it and (neighboring) all its descendant nodes are known to be safe. A pictorial representation of the *inter_non_root* automaton is given in fig 6.

Working together, these automata implement *forest_synch*. As before, we give the formal specifications of the automata described above, prove the implementation correct, and conclude with a few remarks arguing that all fair executions of the implementation generate fair executions of the implemented system.

3.7.1 Root node automata in the intercluster synchronizer

Automaton *inter_root*(p, C), $p = root(C)$, where C is a subtree of *forest*.
We describe the component automaton of *forest_synch* that runs on the root node of every subtree C of *forest*.

Input:
$cluster\_ok(C, i)$
$relay\_cluster\_ok^{inp}(q, p, i)$, $(p, q) \in preferred\_links(forest)$

$ready^{inp}(q,i),\ q \in children(p)$
Output:
$cluster\_go(C,i)$
$relay\_cluster\_ok^{out}(p,q,i),\ q \in children(p)$ *or* $(p,q) \in preferred\_links(forest)$
Internal:
none

State

array $cluster\_ok\_recd(i)$
array $relay\_ok\_recd(q,i)$, $(p,q) \in preferred\_links(forest)$
array $ready\_recd(q,i),\ q \in children(p)$
array $cluster\_go\_sent(i)$
array $relay\_ok\_sent(q,i)$, $(p,q) \in preferred\_links(forest)$

Transition

$cluster\_ok(C,i)$
Effect:
$s.cluster\_ok\_recd(i) = true$

$relay\_cluster\_ok^{inp}(q,p,i)$
Effect:
$s.relay\_ok\_recd(q,i) = true$

$ready^{inp}(q,i)$
Effect:
$s.ready\_recd(q,i) = true$

$cluster\_go(C,i)$
Precondition:
$\forall q$ such that $(p,q) \in preferred\_links(forest)$, $s'.relay\_ok\_recd(q,i) = true$
$\forall q \in children(p)$, $s'.ready\_recd(q,i) = true$
$s'.cluster\_ok\_recd(i) = true$
$s'.cluster\_go\_sent(i) = false$
Effect:
$s.cluster\_go\_sent(i) = true$

$relay\_cluster\_ok^{out}(p,q,i)$
Precondition:
$s'.cluster\_ok\_recd(i) = true$
$s'.relay\_ok\_sent(q,i) = false$
Effect:
$s.relay\_ok\_sent(q,i) = true$

Partitions

All $cluster\_go(C,*)$ actions in one class,
all $relay\_cluster\_ok^{out}(p,q,*)$ actions in one class, for all p and q such that $(p,q) \in preferred\_links(forest)$.

We now describe, in formal terms, the component automaton on each non-root node in a subtree C of $forest$.

3.7.2 Non-root node automata in the intercluster synchronizer

Automaton $inter\_non\_root(p, C)$, $p \in nodes(C)$, $p \neq root(C)$ and C is a subtree of $forest$

Signature

Input:
$relay\_cluster\_ok^{inp}(r, p, i)$, $r = parent(p)$ *or* $(p, r) \in preferred\_links(forest)$
$ready^{inp}(q, i)$, $q \in children(p)$
Output:
$ready^{out}(r, i)$, $r = parent(p)$
$relay\_cluster\_ok^{out}(p, q, i)$, $q \in children(p)$ *or* $(p, q) \in preferred\_links(forest)$
Internal:
none

State

array $cluster\_ok\_recd(i)$
array $relay\_ok\_recd(q, i)$, $(p, q) \in preferred\_links(forest)$
array $ready\_recd(q, i)$, $q \in children(p)$
array $ready\_sent(i)$
array $relay\_ok\_sent(q, i)$, $(p, q) \in preferred\_links(forest$

Transition

$relay\_cluster\_ok^{inp}(r, p, i)$
Effect:
if $r = parent(p)$ then $s.cluster\_ok\_recd(i) = true$
else $s.relay\_ok\_recd(r, i) = true$

$ready^{inp}(q, i)$
Effect:
$s.ready\_recd(q, i) = true$

$ready^{out}(r, i),\ r = parent(p)$
Precondition:
$\forall q$ such that $(p, q) \in preferred\_links(forest)$, $s'.relay\_ok\_recd(q, i) = true$
$\forall q \in children(p)$, $s'.ready\_recd(q, i) = true$
$s'.cluster\_ok\_recd(i) = true$
$s'.ready\_sent(i) = false$
Effect:
$s.ready\_sent(i) = true$

$relay\_cluster\_ok^{out}(p, q, i)$
Precondition:
$s'.cluster\_ok\_recd(i) = true$
$s'.relay\_ok\_sent(q, i) = false$
Effect:
$s.relay\_ok\_sent(q, i) = true$

Partitions

All $ready^{out}(r, *)$ in one class, $r = parent(p)$,
all $relay\_cluster\_ok^{out}(p, q, *)$ actions in one class, for all p and q such that $(p, q) \in preferred\_links(forest)$.

As before, we model asynchronous communication channels by I/O automata, and provide, in the next two subsections, formal specifications of link automata which represent tree edges (*inter_tree_link* automata) and those that represent preferred links (*preferred_link* automata)

3.7.3 Tree link automata

Automaton $Inter\_tree\_link(p \rightharpoonup q, C),\ p = parent(q, C)$.
This automaton allows $inter\_non\_root(p, C)$ or $inter\_root(p, C)$ and $inter\_non\_root(q, C)$ to exchange messages.

Signature

Input:
$ready^{out}(q, i)$
$relay\_cluster\_ok^{out}(p, q, i)$
Output:
$ready^{inp}(q, i)$
$relay\_cluster\_ok^{inp}(p, q, i)$

Internal:
none

State

$buffer$ a fifo queue of any type

Transition

$ready^{out}(q, i)$
Effect:
Add $ready(i)$ to $buffer$

$relay\_cluster\_ok^{out}(p, q, i)$
Effect:
Add $ok(i)$ to $buffer$

$ready^{inp}(q, i)$
Precondition:
first$(s'.buffer) = ready(i)$
Effect:
$s.buffer =$ rest$(s'.buffer)$

$relay\_cluster\_ok^{inp}(p, q, i)$
Precondition:
first$(s'.buffer) = ok(i)$
Effect:
$s.buffer =$ rest$(s'.buffer)$

Partitions

All $ready^{inp}(q, *)$ actions in one class,
all $relay\_cluster\_ok^{inp}(p, q, *)$ actions in one class.

3.7.4 Preferred link intercluster automata

Automaton $Preferred\_link(p \rightharpoonup r, C_p, C_r)$, $p \in C_p, r \in C_r$, C_p *and* C_r distinct trees in $forest$. As mentioned elsewhere, preferred links assist in intercluster communication: they form the communication channels through which each cluster conveys its "*ok*ness" to neighboring clusters.

Input:
$relay\_cluster\_ok^{out}(p, r, i)$
$relay\_cluster\_ok^{out}(r, p, i)$
Output:
$relay\_cluster\_ok^{inp}(p, r, i)$
$relay\_cluster\_ok^{inp}(r, p, i)$
Internal:
none

State

$buffer$ a first-in first-out queue of elements of arbitrary type, initially empty.

Transition

$relay\_cluster\_ok^{out}(p, r, i)$
Effect:
Add $ok(p, r, i)$ to $buffer$

$relay\_cluster\_ok^{out}(r, p, i)$
Effect:
Add $ok(r, p, i)$ to $buffer$

$relay\_cluster\_ok^{inp}(p, r, i)$
Precondition:
$\text{first}(s'.buffer) = ok(p, r, i)$
Effect:
$s.buffer = \text{rest}(s'.buffer)$

$relay\_cluster\_ok^{inp}(r, p, i)$
Precondition:
$\text{first}(s'.buffer) = ok(r, p, i)$
Effect:
$s.buffer = \text{rest}(s'.buffer)$

Partitions

All $relay\_cluster\_ok^{inp}(p, r, *)$ actions in one class,
all $relay\_cluster\_ok^{inp}(r, p, *)$ actions in one class.

Having thus described the distributed implementation of $forest\_synch$, we have now to prove correctness, as before. We again use a possibilities mapping proof to show behavioral inclusion.

3.8 Proof of implementation

Let TER denote the composition of all the $inter\_root$ automata, all the $inter\_non\_root$ automata and all the link automata. Take $\Sigma = \{\pi \in acts(TER) \mid \pi$ is an action that is **not** of the form $cluster\_go(*, *)$ or $cluster\_ok(*, *)\}$ and $\overline{TER} = hide_{\Sigma} TER$. Further, let $\overline{TER}(C, p)$, $p \in nodes(C)$ denote the $inter\_(non)\_root$ automaton at the node p in C.

Theorem 3.8 $\forall s \in states(\overline{TER})$,

$f(s) = \{t \in states(forest\_synch(forest)) \mid \forall C$, *a subtree in* $forest$,
$t.cluster\_ok\_recd(C,i) = s[\overline{TER}(C,root(C))].cluster\_ok\_recd(i)$
$t.cluster\_go\_sent(C,i) = s[\overline{TER}(C,root(C))].cluster\_go\_sent(i)$

Then f *is a* possibilites mapping *from* $\overline{TER}$ *to* $forest\_synch$.

Proof: If $s \in start(\overline{TER})$, the existence of the start state in $forest\_synch$ consistent with f is easily verified. Let s' be a reachable state of $\overline{TER}$, t' any state in $f(s')$ and (s', π, s) a step of $\overline{TER}$. We need to produce an extended step (t', γ, t) of $forest\_synch$ such that $t \in f(s)$ and $\gamma|ext(forest\_synch) = \pi|ext(\overline{TER})$. We distinguish two cases depending on π, and take γ to be π in both cases.

If $\pi = cluster\_ok(C, i)$ for any C, a cluster, the proof of correctness of f is immediate as π is an input action with the right effects in both automata. Otherwise, $\pi = cluster\_go(C, i)$. In this case the only non-trivial argument we have to make is to show that if $cluster\_go$ is enabled in $\overline{TER}$ then the intra-root automaton in all the neighboring clusters have performed the $cluster\_ok$ action for that round.

Consider any neighboring cluster C' such that the preferred link between C and C' is not incident upon either of their root nodes. If C' is not safe, we show that one of $root(C)$'s children will not be ready - to be precise, the child node on the path containing the preferred link between the two clusters will have not sent a *ready* action yet.

We know there exists a sequence of nodes $u_0, u_1, ..., u_n, v_{n'}, v_{n'-1}, .., v_1, v_0$ such that the u_j's form a path from $u_0 = root(C)$ to a leaf of C, the v_j's form a similar path in C' from $v_0 = root(C')$ and the edge $(u_n, v_{n'})$ between the two leaves is a preferred link in G.

If, in state s', $s'[\overline{TER}(C', v_0)].cluster\_ok\_recd(i) \neq true$, then by the implementation of the inter-non-root automata at the descendent nodes, $s'[\overline{TER}(C', v_1)].cluster\_ok\_recd(i) \neq true$. With this as the base case, an inductive argument shows that for all v_j, $0 < j \leq n'$, $s'[\overline{TER}(C', v_j)].cluster\_ok\_recd(i) \neq true$. We exploit this fact to complete the proof.

If $s'[\overline{TER}(C', v_{n'})].cluster\_ok\_recd(i) = false$, then the automaton on the other side of the preferred link (across from $\overline{TER}(C', v_{n'})$) could not have received a $relay\_cluster\_ok$ from the inter-(non-)root automaton at v_n, that is, $s'[\overline{TER}(C', v_n)].relay\_ok\_recd(v_{n'}, i) \neq true$.
Thus $s'.[\overline{TER}(C', v_n)].ready\_sent(i)$ is $false$, as is $s'.[\overline{TER}(C', v_{n-1})].ready\_recd(v_n, i)$. By induction on smaller values of j, for all $j \geq 0$, $s'.[\overline{TER}(C', v_j)].ready\_recd(v_{j+1}, i) = false$. Hence we see that if $s'[\overline{TER}(C', root(C))].ready\_recd(q, i)$ is true for all $q \in children(root(C))$, then all the neighboring clusters considered above will have sent their $cluster_ok$s for that round already.

In the case when the preferred link between C and C' does touch either of their roots, the above proof lends to an easy modification, and we can conclude that if all the $relay\_ok\_recd$ actions and $ready\_recd$ actions are in through the preferred links, then none of the neighboring clusters can be unsafe.

The correctness of f is thus guaranteed, and hence $behs(\overline{TER}) \subseteq behs(forest\_synch)$. ■

From the preceding theorem, we can conclude that TER implements $forest\_synch$ correctly, as both TER and $\overline{TER}$ project identically on the automaton $forest\_synch$. We turn our attention to fairness issues next.

3.9 Fairness of Implementation

The composition automaton TER has the following property: Fair executions of TER (or equivalently, executions that are fair to each of its component automata) generate fair behaviors of $forest\_synch$. We outline below the argument supporting this claim.

Automaton *forest_synch*'s only locally controlled class of actions consists of actions of the form $cluster\_go(C, *)$, where C is a subtree of *forest*.
As before, we need to show that the generated executions of *forest_synch* are fair to each of these classes of actions. It will be shown that unfair behaviors can be generated only from executions of the implementation that are unfair to the automata that generate the *cluster_go*s. We are thus guaranteed that fair behaviors of TER will generate just fair behaviors of *forest_synch*.

Theorem 3.9 *Let β be a fair behavior of TER. Then $\gamma = \beta | forest\_synch$ is a fair behavior of that automaton.*

Proof: Theorem 3.8 guarantees that γ is a behavior of *forest_synch*. Assume that γ is an unfair behavior of *forest_synch*, and let Γ be the unique execution of that automaton which has γ as its behavior. Then there is a subtree C, and a smallest i such that the action $cluster\_go(C, i)$ is enabled in Γ but never performed in either Γ or β.

If $cluster\_go(C, i)$ is enabled in Γ, then, for all subtrees C' such that C' is a neighboring cluster of C or C itself, the action $cluster\_ok(C', i)$ must occur in Γ. Using the fact that β is fair, an inductive argument will show that the occurrence (in β) of each $cluster\_ok(C', i)$ at a neighboring cluster will cause a $relay\_cluster\_ok(p, q, i)$ action to be performed, for some p in $nodes(C')$ and q in $nodes(C)$ (the induction is on the distance of each node from $root(C')$ on the unique path between $root(C')$ and q, the node in C across a preferred-link from a node in C').

We are thus guaranteed that if β is fair, but does not contain $cluster\_go(C, i)$, it is because some *inter_non_root* automaton at one of $root(C)$'s child nodes has not performed a $ready^{out}$ action. A second inductive argument can now be used to show that there exists a sequence of nodes in C, $v_0, v_1, \cdots v_d$, such that $v_0 = root(C)$, $v_{i-1} = parent(v_i)$, for $i = 1, \cdots d$, and v_d is a leaf node of C, with the property that no $ready^{out}(v_j, i)$ occurs in β, for $1 \leq j \leq d$.

But $ready^{out}(v_d, i)$ is enabled in β: the *relay_cluster_ok*s occur in β if there are incident preferred-links onto v_d, and we already know that all the relevant *cluster_ok*s occur in β, thus enabling that action. If β is fair, this action will then occur in that sequence, thus yielding a contradiction.

■

We thus have a node-level implementation of a collection of automata, whose composition synchronizes clients in a provably correct way.

We end this report in the next section with a summary of our efforts and some noteworthy features of this proof of correctness.

4 Summary

In this paper we have offered a formal, rigorous proof of the correctness of Awerbuch's algorithm for network synchronization. We specified both the algorithm and the correctness condition using the I/O automaton model. Since the algorithm uses simpler algorithms for synchronization within

and between 'clusters' of nodes, our proof could have imported as lemmas the correctness of these simpler algorithms, had their correctness been proved elsewhere. Alternatively, the understanding of the modularity that the proof gives us would allow us to see how to safely change the choices of implementation of the separate parts of the synchronizer, independently of one another. Also, we clearly benefit from having carried out the correctness proof in the I/O automaton model which supports modularity, since the network synchronizer is often used as an 'off-the-shelf building block' component in a larger system, and proofs of the correctness of the system will be able to use our proof without change.

References

[1] B. Awerbuch (1985). *Complexity of Network Synchronization,* JACM, Vol. 32, No. 4, October 1985, pp. 804-823.

[2] B. Awerbuch (1985). *Reducing Complexities of Distributed Maximum Flow and Breadth-First Search Algorithms by means of Network Synchronization,* Networks, Vol. 15, pp. 425-437.

[3] N.A. Lynch, M.R. Tuttle (1987). *Hierarchical Correctness Proofs for Distributed Algorithms,* In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing,* pages 137-151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387, 1987.

[4] A. Fekete, N.A. Lynch, L. Shrira (1987). *A Modular Proof of Correctness for a Network Synchronizer,* MIT Technical Report MIT/LCS/TM-341, September 1987.