A Specification and Verification of Intermittent Global Order Broadcast

by

Catherine A. Matlon

Submitted to the Department of Electrical Engineering and Computer Science in parital fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004
[June 2004]

Author ....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by .... [illegible] ....
Nancy A. Lynch
Professor
Thesis Supervisor

Certified by .... [illegible] ....
Roger I. Khazan
Research Staff
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

This work is sponsored by the United States Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

A Specification and Verification of Intermittent Global Order Broadcast

by

Catherine A. Matlon

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Engineering in Computer Science and Engineering

Abstract

The goal of my thesis is to specify, model and verify intermittent global order broadcast. Broadcast means that every process in the system receives a copy of every message. Global order means that all processes deliver the messages in the same order. Intermittent global order means that global order holds during periods of stability and there are no guarantees during periods of instability. A group of processes is stable if each process can communicate with each other process with some minimal quality of service and no process can communicate with another process outside the group.

Intermittent properties, such as intermittent global order, are useful for certain collaborative applications operating in dynamic environments. They help balance conflicting needs for the different applications. We want to be able to formally specify intermittent properties in order to precisely express the guarantees provided by these applications and to be able to verify the algorithms implementing these properties. Because the guarantees hold intermittently, simply stating a definition and building a state-machine specification for an intermittent property is non-trivial. The same is true about verifying that an algorithm satisfies an intermittent property. Existing specification and verification techniques may need to be adjusted.

Acknowledgments

I'm incredibly grateful to Roger Khazan for his patience and insight. We've shared numerous intellectually provoking conversations throughout the course of this project. Special thanks also to Nancy Lynch for introducing me to Roger and helping me find such a rewarding research opportunity.

I'd also like to thank my wonderfully supportive family for all the encouragement they've given me over the years. There's nothing so rewarding at the end of a hard semester as a relaxing visit with a warm and loving family. Thank you for straightening out my priorities when I seemed to lose track of the real world.

Special thanks also to my incredibly tolerant roommate Samantha. I can't believe she's put up with all of my daily idiosyncrasies for so long. It hasn't always been easy, but she's been incredibly helpful and supportive throughout the last five years. Best of luck to her in her new career as we both finally move on to the next big stage in our lives.

Contents

List of Figures

Chapter 1

Introduction

The Information Systems Technology group at MIT Lincoln Labs is currently working on a project informally known as "Robust Chat" (RC) [5]. This project is motivated by the need for robust collaboration systems in dynamic, mission-critical environments. For example, suppose there are several aircrafts flying around as well several separate ground control stations and they would all like to communicate with each other. During a battle, it may be difficult for messages to get through to all recipients. Many communications may be lost or delayed. RC is designed to operate in such an environment. One of the goals of this project is to create a communication system that will balance the needs for global consistency and low latency. Intermittent global order broadcast (IGOB) is one of the properties of the prototype system to achieve this balance.

The RC system is built with several layers. At the top there is a chat server that communicates with remote chat clients. This layer is responsible for the graphical user interface. Beneath that there is a special broadcast server that handles communication with other sites. The broadcast server is composed of ILT and COFIFO algorithms as well as a connection manager. The ILT (Intermittent global order broadcast based on Logical Time) implementation is responsible for broadcasting the messages sent by its clients and ordering the messages received from other clients. This section will be the main focus of this thesis. Next, there is a set of connection-oriented first-in-first-out (COFIFO) channels that enqueue messages for delivery. There is one COFIFO channel for each sender-receiver pair. The COFIFO layer is responsible for ensuring that all messages are received by the process at the other end. This layer also deletes enqueued messages when the process at the other end has been disconnected for an extended period of time. At the same level, there is also a connection manager that is responsible for detecting connection qualities and passing that information on to the COFIFO algorithm and the algorithm for implementing IGOB. This module can be customized by the client to create different standards for different classifications of the connection qualities. Finally, at the bottom, there is a communication channel with a TCP-like protocol used for the actual transmission of messages. Other transfer protocols may be used as long as they can implement a first-in-first-out property. All of this is outlined in Figure 1-1.

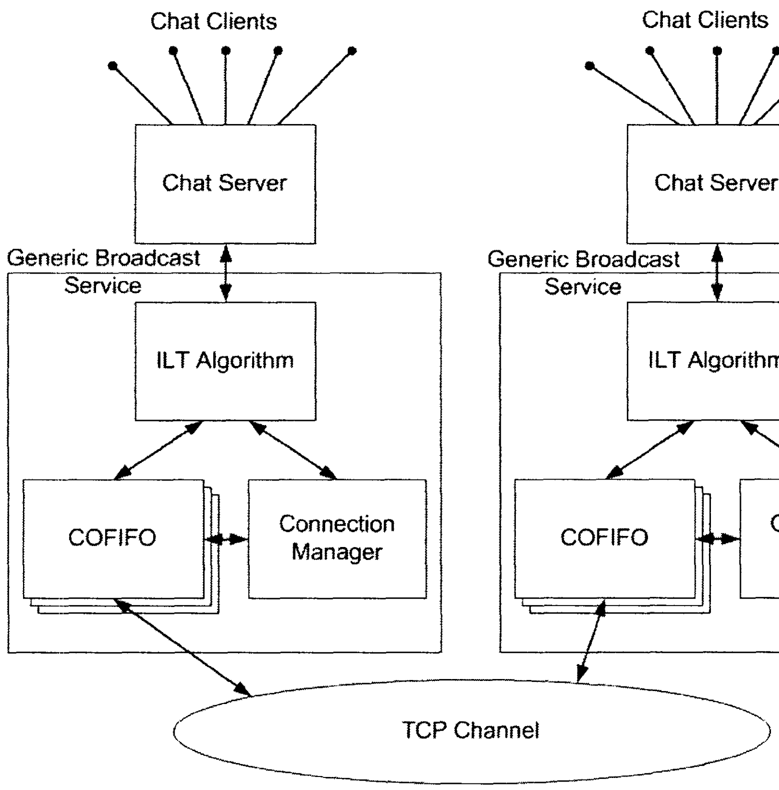

Figure 1-1: A high-level diagram of the Robust Chat System architecture

For my thesis, I am contributing to the RC project by developing the theory for intermittent global order broadcast (IGOB). IGOB is a special property for message ordering in a dynamic environment. Imagine that an airplane and two different ground crews are involved in a discussion in a chat room. The connections between the airplane and the ground may be erratic due to the mobility of planes, attacks, obstructions or other physical factors. Likewise, the communication between the two ground crews may also experience some problems. The three different sites would still like to have a fairly consistent view of the conversation. Specifically, each individual site would like to see the same messages with the same ordering as the other sites. However, if this is a rigid requirement, there may be arbitrarily long message latencies. To illustrate why this may happen, say that Alice and Bob are at the two different ground sites, while Carol is flying around in an airplane. If Carol gets a message through to Bob and then loses contact with Alice, she can't send her message to Alice until she reconnects. However, during this time, Alice wants to be able to receive messages from Bob. But since Bob has already seen Carol's message, Alice needs to wait to receive Carol's message before she can read Bob's messages. Otherwise the order in which Alice views messages would be different from the order in which Bob and Carol view the messages. Depending on the precise algorithm used to implement global order broadcast, different but similar situations may arise. There is no global order broadcast implementation that guarantees perfect ordering without arbitrarily long latencies like the one just presented.

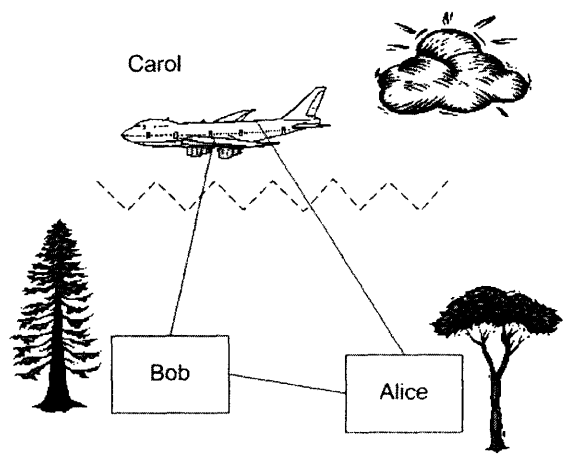

Figure 1-2: Alice, Bob and Carol example

Clearly this is an undesirable situation. Therefore, when the communication channels are unreliable, one would like to loosen the requirements on the message ordering. In the previous example, it would be nice if Alice and Bob could continue to communicate with each other and disregard Carol until she is able to re-establish communication channels with each of them. However, as long as the connections are operating well, one would like to be able to guarantee that Alice, Bob and Carol all see the same messages in the same order. This is roughly how intermittent global order broadcast works. Formally, the specification does not require any guarantees while connections are unstable. But once the connections are stabilized, the different parties must eventually agree on a global ordering for all the messages.

There are other also other methods for weakening global order. The Robust Chat system uses a very lightweight protocol that requires a minimal amount of communication. Therefore, one can only prove that globally consistent ordering will *eventually* hold after all of the connections stabilize. There are other methods that may require additional synchronization. These methods may guarantee global order starting from the end of a synchronization round until a process disconnects.

Although there are many different techniques for handling a dynamic environment, there has been little formal work done in the area of specifying and verifying intermittent properties. Specification is important because it allows us to clearly define what it means to fulfill an intermittent property. The possible limits of a system are much easier to understand once the specification is complete. Verification helps to ensure that a system fulfills a specification. Also, the process of verifying a property typically leads to a greater depth of understanding of how the system behaves. This is because the verification must extend to all possible executions of a system. For example, while verifying intermittent global order, issues of lateness and symmetry arose. Lateness refers to messages that are delivered out of order. In the scenario above, when Carol reconnects and sends her message to Alice, Alice will view that message with a special 'late marking to signal that it was delivered out of order. Connectional symmetry refers to clients at both ends of a channel agreeing on their connection quality. Because the system is asynchronous and distributed, there are situations where Bob may see a brief disconnected from Alice, but Alice does not consider herself disconnected from Bob. Verification of intermittent global order broadcast led to a much fuller understanding of these other issues.

My first step is to create a formal specification for IGOB and model it as a state machine/ automaton. This is a useful computational model that makes it easier for us to reason about which behaviors fulfill the requirements of IGOB. Then, I will model the distributed system developed in the RC project as a composition of state machine components. This places the RC system in the same format as the IGOB specification and allows greater insight into the possible executions. Finally, I will prove that the model of the distributed system satisfies the IGOB specification using a technique known as a simulation proof. Informally speaking, a simulation proof guarantees the the externally visible behavior of the RC system is indistinguishable from the externally visible behavior of the IGOB specification.

Chapter 2

Formal Framework

The techniques used here require all algorithms and properties to be specified with an input/output automaton (IOA). The formal framework for an IOA presented here closely follows chapter 8 of [2]. An automaton A is a special computational structure sometimes also referred to as a state machine. The state of A contains all the values of any variables that may affect the transitions of A. There are also actions ($acts(A)$) the automaton may take in order to transition between the different states. IOA are convenient for modelling distributed systems for a variety of reasons. This model is ideal for asynchronous systems where each component may work at a different speed.

An automaton A consists of five components:

- *sig(A)*, the signature of A, which is a list consisting of all the actions A may take.

- *states(A)*, the set of states of A, which may be infinite.

- *start(A)*, a non-empty subset of states(A), which contains all of the valid start states.

- *trans(A)*, the transitions of A. A transition is a triple (s, π, s') of two states and an action. For every state s and action π, there is a (s, π, s') in trans(A).

- *tasks(A)*, a task partition, which is an equivalence relation on the output and internal actions and breaks them down into countably many equivalence classes.

A transition can be notated as an action name with parameters, preconditions and effects. The preconditions describe in which state the action is legal or *enabled*. For example, say an automaton is designed to control a bank account. If a person wants to withdraw ten dollars from their account, a precondition for the *withdraw(10)* action will require that the account have at least ten dollars in it. The effects describe exactly how the state variables should be altered to reflect the changes made by the action. So using the same example, an effect of *withdraw(10)* would be reducing the value of the account by ten dollars. There are three types of actions: input actions ($in(S)$), output actions ($out(S)$), and internal actions ($int(S)$), where S is the signature of A. So in our banking example, an input action could be a request for account information, an output action would be the ATM displaying the account information, and an internal action would be the bank automatically adding interest to the account at the end of the day. Input actions are externally controlled and must be enabled at all times. For example, a client should always be able to request account information. Internal and output actions are controlled by the automaton. *External* actions are input or output actions, so named because they are visible to an outside observer of the automaton.

An *execution fragment* is a list of alternating states and actions $s_0, \pi_1, s_1, \pi_2, s_2, \ldots$ in which for every k $\geq$ 0, $(s_k, \pi_{k+1}, s_{k+1}) \in$ trans(A). An *execution* is any execution fragment where $s_0 \in$ start(A). Executions may be infinite, but every finite execution must end with a state. A *reachable* state of A is any state that may be the final state of a finite execution of A. The *trace* of an execution of A is the subsequence of the execution that consists only of the external actions. So in the banking example, the trace would show only the requests for account information and the displaying of the account information. Updates from additional interest would not be included.

The equivalence classes of tasks represent different threads of execution. The idea is that during a *fair* execution, each equivalence class gets a fair number of turns to perform a step. If an execution lasts forever, each equivalence class must get an infinite number of chances to perform. With this requirement, we can prove that something will eventually happen. For example, if one automaton sends a message to another automaton, fairness requires that the message will eventually be received.

Different automata can be combined to form a single automaton through a process called *composition*. This can be useful in an environment where we want to use different automata to model different aspects of the system. For example, if Alice and Bob are sending messages to each other, there may be one automaton for Alice, one for Bob, and one for the communication channel in between them. This makes the automata easier to write and understand. In order to compose automata, they must be *compatible*. A set of signatures S_i is compatible if:

- $int(S_i) \cap acts(S_j) = \emptyset \; \forall i \neq j$
- $out(S_i) \cap out(S_j) = \emptyset \; \forall i \neq j$
- No action belongs to infinitely many $acts(S_i)$

Once a set of signatures is compatible, we can define a composition of the corresponding automata with the following definition. The composition $A = \prod_{i \in I}$ is defined as follows:

- $sig(A) = \prod_{i \in I} sig(A_i)$

- $states(A) = \prod_{i \in I} states(A_i)$

- $start(A) = \prod_{i \in I} start(A_i)$

- $trans(A)$ = the set of triples (s, π, s') such that for all $i \in I$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise $s_i = s'_i$

- $tasks(A) = \bigcup_{i \in I} tasks(A_i)$

If there are output actions that we don't want to include in the external trace, there is an option for *hiding* them. We may want to do this if we are composing automata, and the output action of one corresponds to the input action of another. In this case, these external actions are actually being used for internal communication between different components. Thus, we don't want to include these actions in the external trace. Formally, if S is a signature and Φ is an output action, then $hide_{\Phi}(S)$ is the new signature S' where $in(S') = in(S)$, $out(S') = out(S) - \Phi$, and $int(S') = int(S) \cup \Phi$.

An IOA can be used to define a specification. In this situation, the IOA is designed to generate exactly traces that fit the specification. Then, we can show that some other algorithm fulfills the specification by proving that an automaton that models the algorithm implements the specification automaton. Automaton A implements automaton B if all traces generated A are also traces generated by B. So if B is a specification automaton, we know that A can only generate traces that fulfill the specification. Once both the specification and the algorithm are modeled as automata, we can prove that the algorithm implements the specification by using a simulation proof. The simulation proof is based on the idea of *trace inclusion.* Trace inclusion means that the set of traces that may be produced by the algorithm is a subset of the traces that may be produced by the specification.

Generally speaking, a simulation proof works by creating an abstraction that maps states of the algorithm automaton to states of the specification automaton. One must show that this abstraction function is true for all reachable states of the algorithm. Showing that the abstraction function is valid requires two steps. First, one must prove that the initial states of the algorithm map to valid initial states of the specification. Then one must show that each transition in the algorithm corresponds to some sequence of transitions in the specification in such a way that preserves the abstraction function. Furthermore, if a transition π in the algorithm corresponds to the sequence of transtions β in the specification, then both π and β must create the same external trace and β must be enabled whenever π is enabled. Abstractly, it is good to think of a simulation proof as a proof by induction similar to that used in mathematics.

Theorem 2.0.1 *A* **implements** *B in the sense of trace inclusion if there is an abstraction function f that maps states of automaton A to states of automaton B with the following properties:*

1. *If $s \in start(A)$, then $f(s) \cap start(B) \neq 0$*

2. *If s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and $(s, \pi, s') \in trans(A)$, then there is an execution fragment α of B starting with u and ending with some $u' \in f(s')$, such that trace(α)=trace(π)*

During the course of a proof, when reasoning about a transition, the only thing that one can assume is that the transition begins in a reachable state of the automaton. Therefore, the only information that can be used during the simulation proof is information contained in the state. In order to make this viable, it is usually necessary to have some invariants. Invariants are properties of the state that hold in every reachable state of the automaton. Invariants can be proved by induction on the length of the execution sequence. One must show that the invariant holds in the initial states and that, given a reachable state that satisfies the invariant, all transitions from that state preserve the invariant.

Sometimes an automaton discards information that is no longer relevant. Occasionally, it would be nice to still have access to that information in order to prove invariants. For example, say there is a special savings account where no money is withdrawn but interest is compounded monthly. Each month, the automaton deletes the previous value of the account and replaces it with a new value. One might want to prove that the value of the account is always increasing by a larger amount each month. However, proving this requires knowing the previous account values. Therefore, we may introduce what is known as a *history variable.* A history variable [6] is some additional state variable that records information that is not necessary for transitions. Keeping the additional information should not affect the trace of the automaton in any way. Formally, a history variable must satisfy the following constraints:

- Every initial state has at least one value for the history variables.

- No existing transition is disabled by the addition of predicates involving history variables.

- A value assigned to an existing state component does not depend on the value of a history variable.

Chapter 3

Global Order Broadcast

Before we jump into Intermittent Global Order Broadcast, it is important to have an understanding of how message ordering is handled when communication is perfect. Furthermore, the IGOB specification will be based on the GOB Specification. In GOB, there are no disconnections or even temporary loss of communication. In this situation, it is simple to assure total ordering on all messages all of the time. This is referred to as global order broadcast (GOB). Another key motivation for analyzing the case with ideal connections is proof reuse. Proving that an algorithm based on logical time implements intermittent global order broadcast is very similar to proving that a logical time algorithm implements global order broadcast. Later there will be an analysis of the differences between this situation and the dynamic situation.

Suppose Alice is chatting online with a group of friends and they are all in the same chat room. There are some basic properties Alice would like to assume. First, Alice would like to think that all of her friends can see all of the messages she sends. In turn, she would like to see every message sent by her friends. It would also be nice if everybody viewed the messages in the same order. These are some of the properties of global order broadcast.

When a process broadcasts a message, this means that it sends a copy of that message to every process with which it is currently communicating, including itself. In a broadcast system, every process will receive all the messages sent by every other process. There are many different ways to implement a broadcast. One possible algorithm could simply deliver messages to the user in the order that they are received. Another algorithm may hold the messages and deliver them to the user in some special order. This is the case in global order broadcast. A thorough compilation and analysis of different methods for achieving global order broadcast can be found in [1].

Global order broadcast means that every process delivers all the messages in some globally consistent order. This property is vital when users need to have a consistent view of the communications. For example, say that Alice, Bob and Carol are playing a video game where they can shoot at each other and move around. If Alice shoots at Bob and Bob moves at the same time, Bob could either get shot or dodge the bullet depending on the ordering of these two actions. Regardless of what happens, the important thing about the outcome is that Alice, Carol and Bob must all agree on whether Bob was able to escape the bullet or not.

One could easily imagine many other situations where global order is an essential property. Consider a military setting where Alice is trying to count the number of enemy tanks in some region. Say she initially spots five tanks and reports that number back to her commanding officer. Then she spots five additional tanks a few minutes later and reports that there are ten tanks total. Then suppose she realizes the tanks are moving around and some of the new tanks were also part of the initial five she reported, so there are actually only seven tanks total. If different recipients receive her messages in different orders, some may think that the final count is seven tanks while others think there are ten tanks. This is especially bad if there are different protocols to follow based on the number of tanks spotted.

3.1 GOB Specification

3.1.1 Overview

Here we specify global order broadcast with an input-output automaton (IOA). The purpose of this IOA is to make an automaton that may create all possible traces that have the Global Order property. This specification works by placing all messages sent by any process in a GlobalQ. Now when every process retrieves messages by reading them in order off the GlobalQ, it guarantees that every process delivers the same messages in the same order.

3.1.2 GOB Automaton

Automaton GOBroadcastSpec[Ω, M], where Ω is a set of processes and M is a set message alphabet

Define first(queue) to be the first element in the queue

Define queue[i] where i$\in \mathbf{Z}^+$ to be the ith element in the queue

Signature:
Input: send$_p$(m), Process p $\in \Omega$, Message m $\in$ M
Output: deliver$_p$(m), Process p $\in \Omega$, Message m $\in$ M
Internal: order(p), Process p $\in \Omega$

States:
$\forall p \in \Omega$, message queue SendQ$_p$, intially empty
GlobalQ, message queue, intially empty
$\forall p \in \Omega$, integer next$_p$, initially 0

Transitions:

```
input send_p(m)
effects: append m to SendQ_p

internal order(p)
preconditions: first(SendQ_p) ≠ null
effects: append first(SendQ_p) to GlobalQ
    remove first(SendQ_p)

output deliver_p(m)
preconditions: (m) = GlobalQ[next_p]
effects: next_p := next_p+ 1
```

Tasks:
For every p $\in \Omega$, {order(p)}
For every p and q $\in \Omega$, {deliver$_p$(m), m $\in$ M}

3.2 Logical Time Specification

3.2.1 Overview

A fundamental method for ensuring global order is based on logical time due to Lamport [3]. It works by stamping each message with a logical time that creates a total ordering on all messages exchanged within a group of processes. A logical time consists of a pair of numbers. The first number represents the progress of the internal logical clock of a process. A process increments its logical clock every time a message is sent or received so that no two messages sent by the same process will have the same logical clock value. The second number is the unique identification number for that process. Now a total ordering is obtained by sorting messages by the first number and using the second number to break ties.

The logical time algorithm at some process p works by collecting all the messages it receives into a special received_p set. From there, a message m is removed from received_p and delivered to the client when it has the smallest logical time of all the messages in received_p and there is no chance that p will receive a message with a smaller logical time stamp later. Process p knows it will never receive a message with a smaller time stamp when it receives verification from all other processes that their logical clocks have advanced beyond the logical clock value stamped on message m.

In order to improve latency, processes also periodically send out special 'heartbeat messages. These messages simply communicate how far the sender's logical clock has advanced.

This algorithm works under the assumption that processes are connected to each other by first-in-first-out(FIFO) communication channels. This simply means that for any pair of processes, the first process receives the messages sent by the second process in the order that they are sent. Since the logical time algorithm requires all processes to communicate through FIFO channels, we must also specify these channels as IOA. One process transmits messages by placing them on the end of a fifoQ queue, and a second process receives messages by taking them off of the front of the fifoQ queue.

3.2.2 LT_p Automaton

Automaton LT_p[Ω, M], where Ω is a set of processes, p is a process in Ω and M is a set message alphabet.

Let 'heartbeat denote a special character that is not a part of the message alphabet.

T is a set of pairs <n, p> where n $\in \mathbf{N}$ and p $\in \Omega$.

For all t = <n, p> in T, define t+ 1 as <n+1, p>, t.time as n, and t.process as p.

Define t<t' to be true if and only if t.time<t'.time or t.time=t'.time and t.process<t'.process

Signature:
Input: send_p(m), m $\in$ M
    receive_{qp}(<m, t>), Process q $\in \Omega$ - {p}, m $\in$ M $\cup$ {'heartbeat}, t $\in$ T
Output: deliver_p(m), m $\in$ M
    transmit_{pq}(<m, t>), Process q $\in \Omega$ - {p}, m $\in$ M $\cup$ {'heartbeat}, t $\in$ T
Internal: heartbeat_p()
    order_p(m) m $\in$ M

States:
$\forall$ q$\in \Omega$ - {p} pendingQ_p(q), a (M$\cup${'heartbeat}) x T queue, initially empty
$\text{clock}_p \in$ T, initially <0, p>
$\forall q \in \Omega$ - {p}, lt_p(q) $\in$ T, initially <0, q>
received_p, M x T set, initially empty
next_p, an integer, intially 0
GlobalQ_p, message queue, intially empty

Transitions:

```
input sendp(m)
effects: $\text{clock}_p := \text{clock}_p + 1$
    $\forall q \in \Omega$ - {p} append <m, $\text{clock}_p$> to $\text{pendingQ}_p$(q)
    add <m, $\text{clock}_p$> to $\text{received}_p$

output transmitpq(<m, t>)
preconditions: <m, t> = first($\text{pendingQ}_p$(q))
effects: remove <m, t> from $\text{pendingQ}_p$(q)

output deliverp(m)
preconditions: m = $\text{GlobalQ}_p$[$\text{next}_p$]
effects: $\text{next}_p := \text{next}_p + 1$

internal orderp(m)
preconditions: <m, t> $\in$ received
    t.time $\leq$ lt(q).time for all q $\in \Omega$ - {p}
    t $\leq$ t' for all <m', t'> in received
effects: remove <m, t> from received
    append m to $\text{GlobalQ}_p$

input receiveqp(<m, t>)
effects: if m $\neq$ 'heartbeat, add <m, t> to received
    $\text{clock}_p$ := <max[$\text{clock}_p$.time, t.time]+ 1, p>
    lt[q] := t

*internal* $heartbeat_p()$
preconditions: none
effect: $\forall q \in \Omega$ - {p} append<'heartbeat, $clock_p$> to $pendingQ_p(q)$
```

Tasks:
$\forall$ q $\in \Omega$ - {p}, {$transmit_{pq}$(<m, t>) | m $\in$ M, t $\in$ T}
{$deliver_p$(m) | m $\in$ M} $\cup$ {$order_p$(m) | m $\in$ M}
{$heartbeat_p$()}

3.2.3 FIFO Automaton

Automaton FIFO$_{pq}$[A] where A is a set message alphabet and p and q are processes in Ω.

Signature:
Input: transmit$_{pq}$(m) Processes p, q $\in \Omega$, m $\in$ A
Output: receive$_{qp}$(m) Processes p, q $\in \Omega$, m $\in$ A

States:
fifoQ$_{pq}$, a queue of elements of type A, initially empty

Transitions:
input transmit$_{pq}$(m)
effects: append m to fifoQ$_{pq}$

output receive$_{pq}$(m)
preconditions: m = first(fifoQ$_{pq}$)
effects: remove m from fifoQ$_{pq}$

Tasks:
{receive$_{qp}$(m) | m $\in$ A}

3.3 Intermediate Specification

3.3.1 Overview

A formal proof that the logical time algorithm implements global order broadcast is a bit long and tedious, and at times difficult. In the LT specification, there is a special property of all the messages in the GlobalQ$_p$'s. They are sorted in order of increasing logical times. The GlobalQ in the GOB specification allows for almost any arrangement of messages, so there are no nice invariants on the message ordering. This complicates everything because we would like to have nice invariants for the messages ordering that can apply to both automata.

In order to help with that proof, here is an intermediate IOA. The purpose of this automaton is to provide a stepping stone for the proof. Instead of proving that logical time implements global order broadcast directly, we will instead prove that the logical time algorithm implements this intermediate IOA, and the intermediate IOA implements global order broadcast. Then, by transitivity, this proves that logical time implements global order broadcast. With that in mind, this IOA is designed to look like the global order broadcast specification with some of the features of logical time thrown in. Immediately you can see that this algorithm includes the logical clocks at each process. This algorithm also implements the same GlobalQ that was used in the global order broadcast specification, only this time messages are added to the end of the GlobalQ in order of increasing logical time stamps. The ticker transition allows the logical clocks to be randomly incremented so that the algorithm is never stuck.

3.3.2 GBI Automaton

Automaton GlobalBroadcastIntSpec[Ω, M], where Ω is a set of processes and M is a set message alphabet.

```
define getMessage( <m, t> ) as m

define getLogicalTime( <m, t> ) as t

Let T be defined as in the Logical Time IOA.

Signature:

Input: $send_p(m)$, process $p \in \Omega$, message $m \in M$
Output: $deliver_p(m)$, process $p \in \Omega$, message $m \in M$
Internal: order(p), process $p \in \Omega$
    ticker(p), process $p \in \Omega$

States:
$\forall p \in \Omega$, M x T queue, $SendQ_p$, initially empty
$\forall p \in \Omega$, integer $next_p$, intially 0
GlobalQ, a M queue, initially empty
$\forall p \in \Omega$, $clock_p \in T$, initially <0, p>

Transitions:
input $send_p(m)$
effects: $clock_p := clock_p + 1$
    append<m, $clock_p$ > to $SendQ_p$

internal order(p)
preconditions: <m, t> = first($SendQ_p$)
    $\forall q \in \Omega$ - {p}, first($SendQ_q$) $\neq$ null $\Rightarrow$ getLogicalTime(first($SendQ_p$)) <
        getLogicalTime(first($SendQ_q$))
    $\forall q \in \Omega$ $clock_q$.time $\geq$ getLogicalTime(first($SendQ_p$)).time
effects: append getMessage(first($SendQ_p$)) to GlobalQ
remove first($SendQ_p$)

output $deliver_p(m)$
preconditions: m = GlobalQ[$next_p$]
effects: $next_p := next_p + 1$

internal ticker(p)
preconditions: none
effects: $clock_p := clock_p + 1$

Tasks:
$\forall p \in \Omega$, {order(p)},{ $deliver_p(m)$, $m \in M$}, {ticker(p)}
```

3.4 Proof that GBI Spec implements GOB Spec

First, let us define the projection SendQ$|_m$. If SendQ is a M x T queue, then SendQ$|_m$ is a queue that contains only the messages in SendQ.

Lemma 3.4.1 *g is an abstraction function for automaton GBI to automaton GOB.*

Proof 3.4.1 Let us define how the states in GBI map to states in GOB.

g(s $\in$ GBI) = r $\in$ GOB such that:

GOB	GBI
SendQ$_p$()	= SendQ$_p|_m$
next$_p$	= next$_p$
GlobalQ	= GlobalQ

In both Automata, all numerical states are intially 0 and all queues are initially empty. So all that remains to be shown is that for every transition in GBI, there is a corresponding sequence of actions in GOB that preserves the mapping and creates the same trace.

Now we must prove that the abstraction function holds with each transition. Formally, if we are in a reachable state s of GBI and r is the corresponding state of GOB created by the above mapping, then for every action π in GBI, we can find a sequence of actions β in GOB such that the poststate (s') of π in GBI maps to the poststate of β (r') in GOB, or g(s')=r'. Furthermore, π and β must create the same trace and the sequence of actions β must be enabled whenever π is enabled.

Now, let us say that we are in a reachable state s of GBI and g(s) = r.

• π = GBI.send$_p$(m)

The corresponding β in GOB is GOB.send$_p$(m). Both transitions create the same trace, and neither have any preconditions. We need only check that g(s') = r'.

s'.SendQ$_p$ = s.SendQ$_p$ + <m, clock$_p$ >

r'.SendQ$_p$ = r.SendQ$_p$+ m.

s'.clock$_p$ = s.clock$_p$ + 1.

g(s').SendQ$_p$ = g(s.SendQ$_p|_m$ + m) = r.SendQ$_p$ + m = r'.SendQ$_p$.

All other aspects of the state are unaffected by this transition. Since the value of the clock$_p$ variable does not affect the mapping g(), g(s') is the same as r'.

• π = GBI.order(p)

The corresponding β in GOB is GOB.order(p). Neither action affects the trace. The preconditions of GOB.order(p) are strictly contained in the preconditions of GBI.order(p), so GOB.order(p) is enabled whenever GBI.order(p) is enabled. Now we need to check that g(s') is the same as r'.

s'.SendQ$_p$ = s.SendQ$_p$ - first entry.
r'.SendQ$_p$ = r.SendQ$_p$ - first entry.

s'.GlobalQ = s.GlobalQ + m, where m= getMessage(first(s.SendQ$_p$)).
r'.GlobalQ = r.GlobalQ + m, where m=first(r.SendQ$_p$).

g(s').SendQ$_p$ = g(s.SendQ(p)$|_m$ - first entry) = r.SendQ$_p$ - first entry = r'.SendQ$_p$.

g(s').GlobalQ = g(s.GlobalQ + getMessage(first(s.SendQ$_p$)) = r.GlobalQ + first(r.GlobalQ) = r'.GlobalQ.

All other state variables are unaffected by this transition. Hence, g(s') = r'.

• π = GBI.deliver$_p$(m)

The corresponding β in GOB is GOB.deliver$_p$(m). Both actions create the exact same trace. GBI.deliver$_p$(m) is enabled when m=s.GlobalQ[s.next$_p$]. Since r.GlobalQ = s.GlobalQ, s.next$_p$ = r.next$_p$ and GOB.deliver$_p$(m) is enabled when m=r.GlobalQ[r.next$_p$], GOB.deliver$_p$(m) is enabled exactly when GBI.deliver$_p$(m) is enabled. Now we need to check that g(s') = r'.

s'.next$_p$ = s.next$_p$ + 1.
r'.next$_p$ = r.next$_p$+ 1.

g(s') = g(s.next$_p$+ 1) = r.next$_p$ + 1 = r'.next$_p$.

All other state variables are unchanged by this action. Hence, g(s') = r'.

• π = GBI.ticker(p)

The corresponding β in GOB is simply the empty action, λ. Neither of these actions have any effect on the trace, and λ is enabled whenever GBI.ticker(p) is enabled. We need only show that g(s') = r'

s'.clock$_p$ = s.clock$_p$+ 1.

g(s') = g(s) = r = r' since g( ) is unaffected by the clock$_p$ variable.

All other state variables are unchanged. Hence, g(s') = r'

Therefore, since all transitions preserve the mapping, g is an abstraction function from automaton GBI to automaton GOB. ■

Corollary 3.4.1 *GBI implements GOB in the sense of trace inclusion.*

Proof 3.4.1 From Lemma 3.4.1 we have an abstraction function from GBI to GOB, so Theorem 2.0.1 tells us that GBI implements GOB.

3.5 Proof that LT Spec implements GBI Spec

First, we must formally define the automaton LT. Let LT be the composition of Logical Time and FIFO automata, LT = $\prod_{i \in \Omega}$ LT$_i$[Ω, M] $\otimes \prod_{j \in \{\Omega - i\}}$ FIFO$_{ij}$[A], where A = M $\cup$ 'heartbeat. Also, let us treat all trasmit$_{pq}$ and receive$_{qp}$ actions as internal actions and hide them from the external trace.

Let us define a new M x T set LT.Unordered(p, q), where p and q are processes in Ω. Informally, this will be the set of all messages sent by p to q that have not been ordered by any process in Ω. Formally, we construct LT.Unordered(p, q) by appending two queues and a set together and then filtering out irrelevant entries.

For this composition, let us add a history variable to GlobalQ$_p$. Let us assume that GlobalQ$_p$ contains <m, t> pairs instead of just messages.

LT.Unordered(p, q) = LT.pendingQ$_p$(q) $\cup$ fifoQ$_{pq}$ $\cup$ LT.received$_q$ -
{<m, t> such that m='heartbeat} - {<m, t> such that t.process$\neq$q} -
{<m, t> such that <m, t> $\in$ GlobalQ$_r$ for any process r $\in \Omega$}

3.5.1 Lemma and Invariants

Now I will prove a lemma and several invariants that will be necessary when proving the validity of the abstraction function.

Lemma 3.5.1 *There are no non-heartbeat messages in either GB or LT with the same logical time. Or, more formally, for all <m, t> where m $\in$ M that are either sent, transmitted, received, delivered or ordered by any process p $\in \Omega$, there is no other <m', t'> with m $\in$ M that is sent, transmitted, received, delivered or ordered by any process in Ω where t=t'.*

Proof 3.5.1 In order to prove this, consider extensions of both our current Automata with modified send$_p$(m) transitions. Say that each Automaton has a history variable MyMessages$_p$ that contains all the pairs of messages in M and the logical times at which they were created. Modify send$_p$(m) so that it adds <m, clock$_p$ > to MyMessages$_p$ as well as SendQ$_p$ or pendingQ$_p$(q). Now, note that within each set of MyMessages$_p$, no two messages have the same logical time. This is true because clock$_p$ is never decreased by any transition, but it is increased during each send$_p$(m) transition before clock$_p$ is paired with the message and added to MyMessages$_p$. It is also true that logical times cannot be the same for messages created by different processes. This is true because logical time is a pair that includes the process ID, which is unique to each process. So a logical time created by one process can never be the same as a logical time created by another process.

Now realize that any message sent, received, transmitted, delivered, or ordered by any process must be contained in MyMessages$_p$ for some process p in Ω. No message can enter the system unless it is sent by a process, at which point it becomes a member of the MyMessage set of that process. Thus, all the messages encountered in both LT and GBI have unique logical times.

Invariant 3.5.1 *$\forall$ <m, t> $\in$ pendingQ$_p$(q) $\cup$ fifoQ$_{pq}$, t$\leq$ clock$_p$*

Proof 3.5.1 Initially, both queues are empty and clock$_p$ = <0, p>, so the Invariant is satisfied.

LT.send$_p$(m) increments clock$_p$ and appends <m, t> to pendingQ$_p$(q) where t=clock$_p$. Since t$\leq$clock$_p$ this preserves the invahe principal difficulty in converting a proof from the stable case to the intermittent case lies in the invariants. riant.

LT.transmit$_{pq}$(<m, t>) moves <m, t> from pendingQ$_p$(q) to fifoQ$_{pq}$. This also preserves the invariant.

LT.receive$_{pq}$(<m, t>) removes <m, t> from fifoQ$_{pq}$ and increases clock$_p$, which preserves the invariant.

LT.heartbeat$_p$() adds <'heartbeat, clock$_p$> to pendingQ$_p$(q). Since clock$_p$ $\leq$ clock$_p$, this preserves the invariant.

None of the other transitions affect the relevant state variables. Therefore, since all transitions preserve the invariant, the invariant is true in every reachable state.

Invariant 3.5.2 *In every reachable state of LT, LT.clock$_p$ $\geq$ LT.lt$_q$(p) for any pair of processes q and p in Ω such that $q \neq p$*

Proof 3.5.2 Initial state: both LT.clock$_p$ and LT.lt$_q$(p) = 0. Invariant holds.

Transitions: LT.send$_p$(m) increments clock$_p$ and has no effect on LT.lt$_q$(p). So if clock$_p$ $\geq$ LT.lt$_q$(p) before the transition, then it is also true after the transition.

LT.receive$_{qp}$(<m, t>) may increase clock$_p$ and does not change LT.lt$_q$(p) so if the invariant was true before this transition, then it is also true after the transition.

LT.receive$_{pq}$(<m, t>) does not change clock$_p$, but it sets LT.lt$_q$(p) to t. Since t $\leq$ clock$_p$ (from Invariant 1), lt$_q$(p) $\leq$ clock$_p$ after this transition.

No other transitions influence the relevant state variables. Since the invariant holds in the initial state and is preserved by all the transitions, it is true in every reachable state of LT.

Invariant 3.5.3 *All messages from a process p are received by another process q in the order in which they were sent. Formally, when process q receives <m, t> from p, this implies that q will never receive <m', t'> from p where t' < t.*

Proof 3.5.3 It is sufficient to show that fifoQ_{pq} + pendingQ_p(q) is sorted in order of increasing logical times and every time a <m, t> pair enters either fifoQ_{pq} or pendingQ_p(q), there will never be another message <m', t'> with t'<t that enters fifoQ_{pq} or pendingQ_p(q). This is sufficient because all messages received by process q from process p are taken off the front of fifoQ_{pq}.

Initially, both queues are empty, so the invariant holds.

The only time any <m, t> pair can ever enter pendingQ_p(q) is during a heartbeat() or a send$_p$(m) action. Both of these actions pair a message with the current clock$_p$ before adding it to the end of the pendingQ_p(q). Since clock$_p$ is a strictly non-decreasing function, it follows that pendingQ_p(q) is sorted in order of increasing logical times and after <m, t> enters the pendingQ_p(q), no other <m', t'> with t' < t will ever enter pendingQ_p(q).

The only transition that may add <m, t> pairs to fifoQ_{pq} is transmit$_{pq}$(<m, t>). Since this takes messages off the front of pendingQ_p(q) and appends them to the end of fifoQ_{pq}, we can say that fifoQ_{pq} is also sorted in order of increasing ligocal times. Furthermore, after <m, t> enters the fifoQ_{pq}, no other <m', t'> with t' < t will ever enter fifoQ_{pq}.

Invariant 3.5.4 *LT.lt$_p$(q) is always less than or equal to the time stamp of any messages from q that p has not yet received. Or more formally, if <m, t> $\in$ LT.pendingQ$_q$(p)$\cup$ fifoQ$_{qp}$, then t $\geq$ LT.lt$_p$(q).*

Proof 3.5.4 Initially LT.lt$_p$(q) is <0, q>, LT.clock$_q$ is <0, q>, LT.PendingQ_q(p) is empty, and LT.fifoQ_{qp} is empty, so the invariant is satisfied.

The only transition that ever changes LT.lt$_p$(q) is receive$_{qp}$(<m, t>). at this time LT.lt$_p$(q) is changed to t. From invariant 3, we can see that p will never receive a message <m', t'> from q with t' < t. From liveness properties, we know that p must eventually receive all messages in LT.pendingQ_{qp} and LT.fifoQ_{qp}. Therefore, there are never any messages in either LT.pendingQ_q(p) and LT.fifoQ_{qp} with logical time less than LT.lt$_p$(q).

Invariant 3.5.5 *In every reachable state of LT, Unordered(p, q) = Unordered(p, r) for any three processes (not necessarily distinct) p, q, r in Ω*

Proof 3.5.5 Initially, all the queues and sets are empty, so the invariant holds.

Now we need to show that every possible transition preserves the invariant.

LT.send$_p$(m) appends the same message pair <m, clock$_p$> to the pendingQ$_p$(s) of every process s $\neq$ p in Ω, and it also adds <m, clock$_p$> to its received$_p$ set. So now <m, clock$_p$> has been added to Unordered(p, q) for all q $\in \Omega$

LT.transmit$_{pq}$(<m, t>) removes <m, t> from LT.pendingQ$_p$(q) and adds it to LT.fifoQ$_{pq}$. If <m, t> was in Unordered(p, q), this does not remove it. If <m, t> was not in Unordered(p, q) then it is not added.

LT.order$_p$(<m, t>) If p is the first process to add <m, t> to its GlobalQ then it removes <m, t> from Unordered(q, p) for all q and p in Ω. Otherwise, some other process has already placed <m, t> in its GlobalQ and it has already been removed from all Unordered(q, p), so this transaction has no effect.

LT.receive$_{qp}$(<m, t>) simply removes <m, t> from LT.fifoQ$_{qp}$ and adds it to LT.received$_q$. If <m, t> was in Unordered(p, q), this does not remove it. If <m, t> was not in Unordered(p, q) then it is not added.

LT.heartbeat$_p$() does not affect Unordered(p, q) for any q in Ω because the definition of Unordered(p, q) does not allow it to include any heartbeat messages.

No other transitions affect any of the queues or sets involved in the definition of Unordered(p, q).

Hence, in every reachable state of LT, Unordered(p, q) and Unordered (p, r) are the same for all p, q, r $\in \Omega$

Invariant 3.5.6 *All of the LT.GlobalQ's are prefixes of each other. Formally, for all LT.GlobalQ$_p$ and LT.GlobalQ$_q$, either LT.GlobalQ$_p$ is a prefix of LT.GlobalQ$_q$ or LT.GlobalQ$_q$ is a prefix of LT.GlobalQ$_p$.*

Proof 3.5.6 For this proof, let us consider an extension of LT where the GlobalQ contains message and logical time pairs rather than just messages. Say that order$_p$ <m, t> appends <m, t> to GlobalQ$_p$ instead of just m. I will first prove that each LT.GlobalQ contains M x T pairs sorted by increasing values of t.

Proof that LT.GlobalQ$_p$ is sorted by increasing logical time values: LT.GlobalQ$_p$ is altered only by the order$_p$(<m, t>) transition. This will only append <m, t> to GlobalQ$_p$ if t $\leq$ t' for all <m', t'> in received$_p$ and if t.time $\leq$ lt$_p$(q) for all q in Ω-{p}. From invariant 4 we know that we can never receive any messages from q with logical time less than or equal to lt$_p$(q). So if t.time $\leq$ lt$_p$(q) for all q, then we will never receive any messages <m', t'> with t' < t. If <m, t> has the smallest t for all <m', t'> in received and we can never receive any message with a smaller logical time, then we can conclude that LT.GlobalQ$_p$ is sorted by increasing values of t. Also, because we can never receive any messages with a smaller logical time than t and liveness tells us we eventually receive all messages, this implies that there are no missing messages in any of the GlobalQ's. That is, if some GlobalQ contains message <m, t>, then it must also contain all <m', t'> with t' < t. Therefore, all of the GlobalQ's must be prefixes of each other.

3.5.2 Mapping and Simulation

Lemma 3.5.2 *f is an abstraction function from automaton LT to automaton GBI.*

Proof 3.5.2 First, let's go define the mapping f(), which maps states in GlobalBroadcastIntSpec to states in LT.

f(s ∈ LT) = r ∈ GBI such that:

GBI	LT
SendQ$_p$	= Unordered(p, q) sorted by increasing logical times
next$_p$	= next$_p$
GlobalQ	= one of the longest GlobalQ$_p$ of all processes p in Ω
clock$_p$	= clock$_p$

In both Automata, all numerical states are initially 0 and all queues are initially empty. All clock values are initially <0, p> in both Automata. Hence they both have the same initial state. We need only to show that for every transition in LT there is a corresponding sequence of transitions in GBI that preserves the mapping and creates the same trace.

Formally, we must show that if we are in a reachable state s of LT and r is the corresponding state of GBI created by the above mapping, then for every action π in LT, we can find a sequence of action β in GBI such that the poststate (s') of π in LT maps to the poststate of β (r') in GBI, or f(s')=r'. Furthermore, π and β must create the same trace and the sequence of actions β must be enabled whenever π is enabled.

Now let us say that we are in a reachable state s of LT and f(s)=r

- π = LT.send$_p$(m)

The corresponding β is GBI.send$_p$(m). Both transitions create the same trace, and both are always enabled. We need only check that f(s')=r'.

s'.clock$_p$ = s.clock$_p$+ 1.

r'.clock$_p$ = r.clock$_p$+ 1.

For all q $\in \Omega$ such that q $\neq$ p, s'.pendingQ$_p$(q) = s.pendingQ$_p$(q) + <m, s.clock$_p$ + 1 >. (Note that for all q $\in \Omega$ such that p $\neq$ q, s'.Unordered(p, q) = s.Unordered(p, q) $\cup$ {<m, s.clock$_p$ >+ 1})

s'.received$_p$ = s.received$_p$ $\cup$ {<m, s.clock$_p$ + 1 >}. (Note that
s'.Undelievered(p, p) = s.Unordered(p, p) $\cup$ {<m, s.clock$_p$+ 1>})

s'.Unordered(p, q) = s.Unordered(p, q) $\cup$ {<m, s.clock$_p$ + 1 >} for all q $\in \Omega$).

r'.sendQ$_p$ = r.sendQ$_p$ + <m, r.clock$_p$ + 1 >.

f(s'.clock$_p$) = f(s.clock$_p$+ 1) = r.clock$_p$ + 1 = r.clock$_p$

f(s'.Unordered(p, q)) = f(s.Unordered(p, q) $\cup$ {<m, s.clock$_p$+1 >}) = r.sendQ$_p$ + <m, r.clock$_p$+ 1 > = r'.sendQ$_p$.

f(s.Unordered(p,q)) is defined as a set with no ordering, and r.sendQ$_p$ does have order. Because GBI.send$_p$(m) appends <m, r.clock$_p$ + 1 > to the end of r.SendQ$_p$, we must confirm that <m, r.clock$_p$ + 1 > has the largest logical time of all elements of s'.Undelivered(p, q). This is true because the clock$_p$ is a strictly nondecreasing variable and <m, s.clock$_p$+ 1> is the most recent addition to Unordered(p, q). Therefore <m, s.clock$_p$ + 1 > must have the largest logical time of all M x T pairs in Unordered(p, q).

All other state variables remain unchanged. Hence f(s') = r'.

- π = LT.transmit$_{pq}$(<m, t>)

The corresponding β is simply the empty transition λ. Since λ is always enabled, we know that β is enabled whenever π is enabled. Since we will hide all transmit and receive actions in the final trace, neither action here will impact the final trace. Now we need only show that f(s') = r'.

s'.pendingQ$_p$(q) = s.pendingQ$_p$(q) - <m, t>

s'.fifoQ_{pq} = s.fifoQ_{pq} + <m, t>.

From the definition, we can see that this change does not affect Unordered(p, q). Therefore f(s'.Unordered(p, q)) = f(s.Unordered(p, q)) = r.SendQ_p = r'.SendQ_p. Since all other state variables are unaffected by this transition, we can conclude that f(s') = r'.

• π = LT.deliver$_p$(m)

The corresponding β is GBI.deliver$_p$(m). (Note: since the LT.GlobalQ_p are all prefixes of each other, and GBI.GlobalQ = one of the longest LT.GlobalQ_p of all p $\in \Omega$, it is safe to say that LT.GlobalQ_p is a prefix of GBI.GlobalQ for any p $\in \Omega$) Since LT.GlobalQ_p is a prefix of GBI.GlobalQ, if m = LT.GlobalQ_p[s.next$_p$] then m = GBI.GlobalQ[r.next$_p$]. Therefore, GBI.deliver$_p$(m) is enabled whenever LT.deliver$_p$(m) is enabled. They both create the same external trace, so all we need to check now is that f(s') = r'.

s'.next$_p$ = s.next$_p$+ 1

r'.next$_p$ = r.next$_p$+ 1

f(s'.next$_p$) = f(s.next$_p$+ 1) = r.next$_p$+ 1 = r'.next$_p$.

All other state variables are not affected by this transition. Hence, f(s') = r'.

• π = LT.order$_p$(m), and let <m, t> be the message-logical time pair that satisfy the first precondition.

The corresponding β can be one of two actions:

1. if $\forall$ q $\in \Omega$, m$\notin$ GlobalQ_q, then β = GBI.order(t.process).

2. Otherwise, $\exists q \in \Omega$ such that q has already performed LT.order$_q$(m), so β is simply the empty action λ.

I will start with case 2, since it is the simpler one to prove. Since λ is always enabled, we know that β is enabled whenever LT.order$_p$(m) is enabled. And since LT.order$_p$ is an internal action, the trace is the same in both cases. Now all that remains is to prove that f(s') = r'

s'.received = s.received - <m, t>.

Note that since some other process q has already performed LT.order$_q$(m), we know from the definition of Unordered that <m, t> is not in Unordered(t.process, r) for any process r $\in \Omega$, so Unordered(t.process, q) is unchanged for all q $\in \Omega$. Hence, the mapping to GBI.Send$Q_{t.process}$ is unchanged.

s'.GlobalQ$_p$ = s.GlobalQ$_p$ + <m, t>.

Note that some other process q has already performed order$_q$(m), so s.GlobalQ$_q$ already included <m, t>. Since all GlobalQ's are prefixes of each other, it follows that s.GlobalQ$_q$ is strictly larger than s.GlobalQ$_p$. Therefore, when we add a single message to s.GlobalQ$_p$, s'.GlobalQ$_p$ is at most as long as s.GlobalQ$_q$. Therefore, s'.GlobalQ$_p$ is not longer than one of the longest GlobalQ's in s. From invariant 3.5.6, we know that s'.GlobalQ$_p$ is a prefix of one of the longest GlobalQ's in s. Hence, the mapping to GBI.GlobalQ remains unchanged. Since λ also does not affect the mapping, it follows that f(s') = r'.

In the first case, we know that p is the first process to perform LT.order$_p$ on message <m, t> and the corresponding β is GBI.order(t.process). Since both are internal actions, they do not affect the external trace. We need to show that GBI.order(t.process) is enabled whenever LT.order$_p$ <m, t> is enabled.

The first precondition for GBI.order(t.process) is that <m, t> = first(SendQ$_{t.process}$). From the definition of Unordered(p, q) we can see that in state s, <m, t> is in Unordered(t.process, p) for all p $\in \Omega$. We need only show that for all <m', t'> in Unordered(t.process, p), t < t'. From invariant 3.5.4, we know that if t.process sent any earlier messages, they must have been received by p. Hence, any <m', t'> with t' < t must be in either LT.received$_p$ or LT.GlobalQ$_p$. If it is in LT.GlobalQ$_p$, then we know it cannot be in Unordered(t.process, p). If it is in LT.received$_p$, then received$_p$ has a message with a time stamp less than t, which contradicts the third precondition. Hence, <m, t>=r.first(SendQ$_{t.process}$) whenever LT.order$_p$ <m, t> is enabled.

With invariant 3.5.2 (LT.lt$_q$(p) $\leq$ LT.clock$_p$), it follows that the third precondition of GBI.order(t.process) is satisfied whenever the second precondition of LT.order$_p$(m) is satisfied.

From invariant 3.5.4 and the definition of Unordered(p, q), we can see that either r.first(SendQ$_q$) is in s.received$_p$ or s.lt$_p$(q) < t. If s.lt$_p$(q) < t then π is not enabled. If r.first(SendQ$_q$) is in s.received$_p$ then the third precondition of LT.order$_p$ <m, t> implies that the second precondition of GBI.order(t.process) is satisfied.

Hence, β is enabled whenever π is enabled.

Now we must show that f(s') = r'.

s'.received$_p$ = s.received$_p$ - <m, t>.

s'.GlobalQ$_p$ = s.GlobalQ$_p$ + m.

s'.Unordered(t.process, q) = s.Unordered(t.process, q) - <m, t>.

r'.SendQ$_{t.process}$ = r.SendQ$_{t.process}$ - <m, t>. r'.GlobalQ = r.GlobalQ + m.

So f(s'.Unordered(t.process, q)) = r'.SendQ$_{t.process}$ for any q $\in \Omega$. Because p is the first process to append m to its GlobalQ, and all of the GlobalQ's are prefixes of each other, we can conclude that s.GlobalQ$_p$ is not shorter than s.GlobalQ$_q$ for all q $\in \Omega$ where q $\neq$ p, and after appending m, GlobalQ$_p$ is the longest GlobalQ. Therefore, s.GlobalQ$_p$ = r.GlobalQ and s'.GlobalQ$_p$ = r'.GlobalQ$_p$. Since no other state variables change, we can conclude that f(s') = r'

- π = LT.receive$_{qp}$(<m, t>)

The corresponding β is simply ticker(p) performed (max[t.time - s.clock$_p$, 0] + 1) times. For convenience, we will simply refer to this number as n in the next few paragraphs. Since GBI.ticker(p) is always enabled, we know that β is enabled whenever π is enabled. Since we will hide all transmit and receive actions in the final trace and GBI.ticker(p) is an internal action, neither will impact the final trace. Now we need only show that f(s') = r'.

s'.fifoQ$_{pq}$ = s.fifoQ$_{pq}$ - <m, t>

s'.received$_q$ = s.received$_q$ + <m, t>

From the definition of Unordered(p, q), we can see that this change does not affect Unordered(p, q). So f(s'.Unordered(p, q)) = f(s.Unordered(p, q)) = r.SendQ$_p$ = r'.SendQ$_p$.

s'.clock$_p$ = s.clock$_p$ + n

r'.clock$_p$ = r.clock$_p$ + n

f(s'.clock$_p$) = f(s.clock$_p$ + n) = r.clock$_p$ + n = r'.clock$_p$.

s.lt$_p$(t.process) is also changed, but this does not affect any of the states in GBI. Since all other state variables remain unchanged, we can conclude that f(s') = r'.

- π = LT.heartbeat$_p$()

The corresponding β in GBI is λ. λ is always enabled. LT.heartbeat$_p$() is an internal action, neither affect the external trace. Now we need only prove that f(s') = r'.

s'.pendingQ$_p$(q) = s.pendingQ$_p$(q) + <'heartbeat, s.clock$_p$ > for all q $\neq$ p.

Note that since all 'heartbeat messages are not a part of Unordered(p, q), we can see that Unordered(p, q) is unaffected by this transition, for any p and q in Ω. Therefore, the mapping is not influenced and f(s') = r'.

Hence, f is an abstraction function from automaton LT to automaton GBI. ■

Corollary 3.5.1 *LT implements GBI in the sense of trace inclusion. Furthermore, by transitivity, LT implements GOB.*

Proof 3.5.1 From lemma 3.5.2 we know that there is an abstraction function from LT to GBI. Therefore, by Theorem 2.0.1, LT implements GBI in the sense of trace inclusion. And by transitivity and Corollary 3.4.1, we can see that LT implements GOB.

Chapter 4

Intermittent Global Order Broadcast

Now imagine that a process is in a dynamic environment where any processes can enter or leave the group (or disconnect and reconnect) at any time. With global order broadcast, one process cannot view messages until all processes somehow agree on the order. So in this dynamic environment, it is impossible to perfectly implement global order broadcast without allowing for arbitrarily long message latency. Therefore, in order to bound message latency and increase availability of the broadcast one must weaken the requirements on the message ordering.

There are many different scenarios where one could be concerned with group communication in a dynamic environment. The RC project, for example, is designed for communication between various ground control bases and aircrafts. In a mobile environment with wireless connections, the communication between ground control and an airplane or between two airplanes may fail frequently. Ideally, a single airplane with erratic connections should not slow down the communications of the rest of the group beyond a certain point.

There are a number of different approaches to this problem. One approach is to simply guarantee that all messages from a particular process are received in the order in which they were sent, without requiring that the messages be globally ordered. Another approach is to simply provide a best-effort global order broadcast algorithm without any strong assurances. One could also guess the ordering of the messages and then correct it later if there are mistakes. Yet another possibility is to have a special round of communication after someone is reconnected. This round would be used to update everyone's information so that global order within that subgroup could begin immediately. While this guarantees global order broadcast as soon as possible after stability is reached, it also requires additional rounds of communication, which is especially bad if there are frequent changes in connectivity.

The approach taken in the RC project is different. It uses a light-weight approach that does not include any special synchronization steps. Processes continue to send and receive messages using the same protocol they used in the past, and global order will eventually start to hold during a period of stability. Informally, a period of stability is a length of time when no processes enter or leave the group and all the communication channels provide some minimal quality of service. To prevent unbounded latencies, there are looser guarantees when the environment is unstable. If a subset of processes always have good communication between each other, they will view each other's messages in a globally consistent order. The only messages they may disagree on are messages from outside processes with faulty communication lines.

4.1 Formal Definition

Before formally defining intermittent global order, it is necessary to define a few other properties first. To do this, we need to introduce the idea of a connection quality. In the RC system, we use three different connection qualities designed to indicate how well the communication channel is performing. The connection qualities are 'connected', 'suspected', and 'disconnected'. A quality level of 'connected' means that the connection is active, and the quality is above some minimal level. A 'disconnected' quality represents a connection that is down. A 'suspected' quality may either reflect a connection that is operating below some minimal quality level or a connection that has recently been severed. The exact service requirements for each of the connection qualities may be customized by the user.

In order to define everything formally, we must define the external interface of the system. Our definitions will apply to an automaton that accepts Ω (a set of processes) and M (a set message alphabet) as variables and has an external signature consisting of the following actions:

- send$_{pq}$(m) where p, q are processes in Ω and m $\in$ M
- receive$_{qp}$(m) where p, q are processes in Ω and m $\in$ M
- statusUpdate$_{pq}$(quality) where p, q are processes in Ω and
  m $\in$ M and quality$\in$'connected', 'suspected', 'disconnected'

Definition 4.1.1 *At some point in a trace, two processes p and q in Ω are 'connected' if the last statusUpdate$_{pq}$ and the last statusUpdate$_{qp}$ both used the argument 'connected', and no further statusUpdate$_{pq}$ or statusUpdate$_{qp}$ are ever received with a different argument.*

Definition 4.1.2 *AT some point in a trace, two processes p and q in Ω are 'disconnected' if the last statusUpdate$_{pq}$ and the last statusUpdate$_{qp}$ both used the argument 'disconnected' and no further*

$statusUpdate_{pq}$ or $statusUpdate_{qp}$ are ever received with a different argument.

Definition 4.1.3 *A subset S of processes is considered* **connected** *if $\forall p$ and $q \in S$, p and q are 'connected'.*

Definition 4.1.4 *A set of processes Ω is* **stable** *when $\forall p \in \Omega$ and $\forall q \in \Omega - \{p\}$, the last $statusUpdate_{pq}$ in the trace was $statusUpdate_{pq}$('connected') and $\forall r \notin \Omega$ the last status $update_{pr}$ is $statusUpdate_{pr}$('disconnected') and no later statusUpdates are received that change a connection quality. Stability officially begins in the state immediately following the statusUpdate that makes these conditions true.*

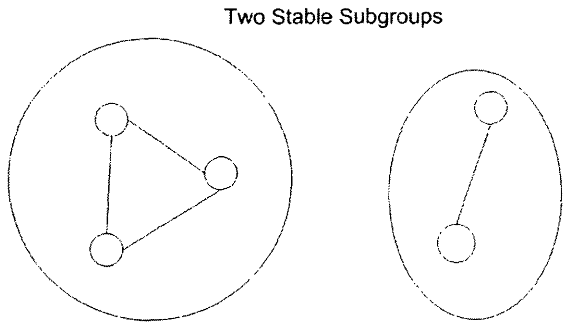

Figure 4-1: Stability Example

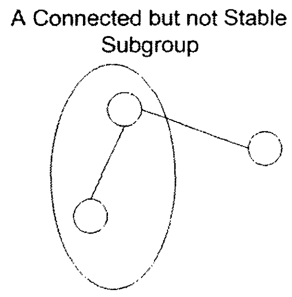

Figure 4-2: Connected Example

In this definition, we require that stability last forever. While this is important in the theoretical setting, it does not seem to make much sense in the practical world. We developed intermittent global order broadcast because there environment is dynamic. The proof actually only requires that stability hold long enough for all of the 'old' messages to be delivered. However, because there is no way to specify time in this system, the only way to guarantee that stability lasts long enough is to force it to hold forever. Then, we can use liveness properties to guarantee that all 'old' messages are delivered. More infomally, it is okay to assume that stability lasts forever because the assumption requires the system to try to implement intermittent global order during a temporary period of stability. This is because when stability begins, there is no way to tell how long it will last.

Say a system implements intermittent global order broadcast if after achieving stability, it eventually generates a trace that is indistinguishable from the trace of the Intermittent Global Order Broadcast Specification(IGOB), which follows on the next page. Formally, a system implements IGOB if after achieving stability, there is some point in the execution where the tail of the trace may be generated by the IGOB specification. A subset of processes may implement intermittent global order broadcast if after it stabilizes, the projection of the trace that includes only transitions involving the subset of processes eventually becomes indistinguishable from a trace of the IGOB specification. That is, there is a point in the execution where the tail of the trace may also be generated by the IGOB specification.

I will prove that our implementation ILT, which is specified in the following pages, implements IGOB as described above. For notational purposes, let us say that messages sent before stability is reached are 'old', and messages sent after stability is reached are 'new'. The trace of ILT is indistinguishable from the trace for IGOB starting immediately after all 'old' messages have either been lost or delivered to their destinations.

4.2 Specification

4.2.1 Overview

This IOA is almost identical to the IOA for global order broadcast that was introduced earlier. In fact, the only difference is the initial state of the algorithm. This is necessary to capture the idea that this IOA begins in the middle of a conversation where different processes have already sent and received messages.

4.2.2 IGOB Automaton

Automaton IGOBroadcastSpec[Ω, M, GlobalQ$_{init}$, next$_{p-init}$, SendQ$_{p-init}$], where Ω is a set of processes and M is a set message alphabet, GlobalQ$_{init}$ is a queue of messages m $\in$ M, next$_{p-init}$ $\in \mathbf{Z}^+$, SendQ$_{p-init}$ is a queue of messages m $\in$ M

Define first(queue) to be the first element in the queue

Define queue[i] where i$\in \mathbf{Z}^+$ to be the ith element in the queue

Signature:
Input: send$_p$(m), Process p $\in \Omega$, Message m $\in$ M
Output: deliver$_p$(m), Process p $\in \Omega$, Message m $\in$ M
Internal: order(p), Process p $\in \Omega$

States:
$\forall p \in \Omega$, SendQ$_p$, a M queue, intially SendQ$_{p-init}$
GlobalQ, a M queue, intially GlobalQ$_{init}$
$\forall p \in \Omega$, integer next$_p$, initially next$_{p-init}$

Transitions:
input send$_p$(m)
effects: append m to SendQ$_p$

internal order(p)
preconditions: first(SendQ$_p$) $\neq$ null
effects: append first(SendQ$_p$) to GlobalQ
    remove first(SendQ$_p$)

output deliver$_p$(m)
preconditions: m = GlobalQ[next$_p$]
effects: next$_p$ = next$_p$+ 1

Tasks:
For every p $\in \Omega$, {order(p)}; {deliver$_p$(m), m $\in$ M}

4.3 Intermittent Logical Time Specification

4.3.1 Overview

Now I will provide an IOA that models the intermittent global order broadcast implementation used in the RC project, which is designed to closely model the logical time algorithm. RC has a few special features. This IOA is set up to handle inputs from a connection manager that provides updates on the connection qualities. The specifics of the connection manager and its affect on performance will be discussed later.

This implementation sends messages to all processes with 'connected' or 'suspected' connection qualities. It only waits on advanced logical clocks from 'connected' processes before ordering and delivering messages. There is also a special marker_p that is used to record the latest logical time of all ordered messages. This marker_p is used to detect whether messages are late or not.

Because we are trying to model a dynamic environment, we have to allow for connections to break down. A connection-oriented FIFO channel is designed to model a point-to-point protocol that uses a TCP connection and tries to re-establish new TCP connections when old ones fail. While a TCP connection is alive, all messages are guaranteed to be delivered in the order that they were sent. When a TCP connection dies, the enqueued messages are lost. We model this with a $lose_{pq}$ transition that allows the channel to drop the messages from the end of the fifoQ queue. And while the quality is 'disconnected', messages can't get through.

4.3.2 ILT$_p$ Automaton

Automaton ILT$_p$[Ω, M], where Ω is a set of processes, p is a process in Ω and M is a set message alphabet.

Let 'heartbeat and 'late denote special characters that are not a part of the message alphabet.

T is a set of pairs <n, p> where $n \in \mathbf{N}$ and $p \in \Omega$.

For all t = <n, p> in T, define t+ 1 as <n+1, p>, t.time as n, and t.process as p.

Define t<t' to be true if and only if t.time<t'.time or t.time=t'.time and t.process<t'.process

Signature:

Input: send$_p$(m), m ∈ M
  receive$_{qp}$(<m, t>), Process q ∈ Ω - {p}, m ∈ M ∪ {'heartbeat}, t ∈ T
  statusUpdate$_{pq}$(quality), Process q ∈ Ω - {p}, quality ∈ {'connected', 'suspected', 'disconnected'}

Output: deliver$_p$(m), m ∈ M
  transmit$_{pq}$(<m, t>), Process q ∈ Ω - {p}, m ∈ M ∪ {'heartbeat}, t ∈ T

Internal: heartbeat$_p$()
  order$_p$(m) m ∈ M

States:

∀ q∈ Ω - {p} pendingQ$_p$(q), a (M∪{'heartbeat}) x T queue, initially empty
clock$_p$ ∈ T, initially <0, p>
$\forall q \in \Omega$ - {p}, lt$_p$(q) ∈ T, initially <0, q>
received$_p$, a M x T set, initially empty
next$_p$, an integer, intially 0
GlobalQ$_p$, a {M ∪ 'late} queue, intially empty
∀ q∈ Ω - {p}, quality$_{pq}$ ∈ {'connected', 'suspected', 'disconnected'}, initially 'disconnected'
suspConn$_p$, a Ω set, initially empty
goodConn$_p$, a Ω set, initially empty
marker$_p$ ∈ T, initially <0, 0>

Transitions:

```
input send_p(m)
effects: clock_p := clock_p + 1
    ∀q ∈ goodConn_p ∪ suspConn_p, append <m, clock_p> to pendingQ_p(q)
    add <m, clock_p> to received_p

output transmit_pq(<m, t>)
preconditions: <m, t> = first(pendingQ_p(q))
effects: remove <m, t> from pendingQ_p(q)

output deliver_p(m)
preconditions: m = GlobalQ_p[next_p]
effects: next_p := next_p+ 1

internal order_p(m)
preconditions: <m, t> ∈ received
    t.time ≤ lt(q).time for all q ∈ goodConn_p
    $t \leq t'$ for all <m', t'> in received
effects: remove <m, t> from received
    if t>$\text{marker}_p$ append m to $\text{GlobalQ}_p$
    if t$\leq$ $\text{marker}_p$ append (m + 'late) to $\text{GlobalQ}_p$
    $\text{marker}_p$ = max [$\text{marker}_p$, t]

*input receive$_{qp}$(<m, t>)*
effects: if m$\neq$'heartbeat, add <m, t> to received
    $\text{clock}_p$ := <max[$\text{clock}_p$.time, t.time]+ 1, p>
    lt[q] := t

*internal heartbeat$_p$()*
preconditions: none
effect: $\forall q \in$ $\text{goodConn}_p$ $\cup$ $\text{suspConn}_p$, append<'heartbeat, $\text{clock}_p$> to $\text{pendingQ}_p$(q)

*input statusUpdate$_{pq}$(quality)*
effects: $\text{quality}_{pq}$ = quality
    if quality = 'disconnected' remove q from $\text{suspConn}_p$ and $\text{goodConn}_p$
    if quality = 'connected' remove q from $\text{suspConn}_p$ and add q to $\text{goodConn}_p$
    if quality = 'suspected' remove q from $\text{goodConn}_p$ and add q to $\text{goodConn}_p$
```

Tasks:

$\forall$ q $\in \Omega$ - {p}, {transmit_{pq}(<m, t>) | m $\in$ M, t $\in$ T}
{deliver_p(m) | m $\in$ M} $\cup$ {order_p(m) | m $\in$ M}
{heartbeat_p()}

4.3.3 COFIFO Automaton

Automaton COFIFO$_{pq}$[A] where A is a set message alphabet and p and q are processes in Ω.

Signature:
Input: transmit$_{pq}$(m) Processes p, q $\in \Omega$, m $\in$ A
statusUpdate$_{pq}$(quality) Processes p, q $\in \Omega$, quality $\in${'connected', 'suspected', 'disconnected'}
statusUpdate$_{qp}$(quality) Processes p, q $\in \Omega$, quality $\in${'connected', 'suspected', 'disconnected'}
Output: receive$_{qp}$(m) Processes p, q $\in \Omega$, m $\in$ A
Internal: lose$_{pq}$ Processes p, q $\in \Omega$

States:
fifoQ$_{pq}$, a {A} queue, initially empty
quality$_{pq}$ $\in$ {'connected', 'suspected', 'disconnected'}, intially 'disconnected'
quality$_{qp}$ $\in$ {'connected', 'suspected', 'disconnected'}, intially 'disconnected'

Transitions:
input transmit$_{pq}$(m)
effects: append m to fifoQ$_{pq}$

output receive$_{pq}$(m)
preconditions: m = first(fifoQ$_{pq}$)
quality$_{qp}$ $\neq$ 'disconnected'
effects: remove m from fifoQ$_{pq}$

input statusUpdate$_{pq}$(quality)
effects: quality$_{pq}$ = quality

input statusUpdate$_{qp}$(quality)
effects: quality$_{qp}$ = quality

internal lose$_{pq}$
preconditions: quality$_{pq}$ = 'disconnected'
effects: remove message on end of fifoQ$_{pq}$

Tasks:
{receive$_{pq}$(m) | m $\in$ A}; {lose$_{pq}$}

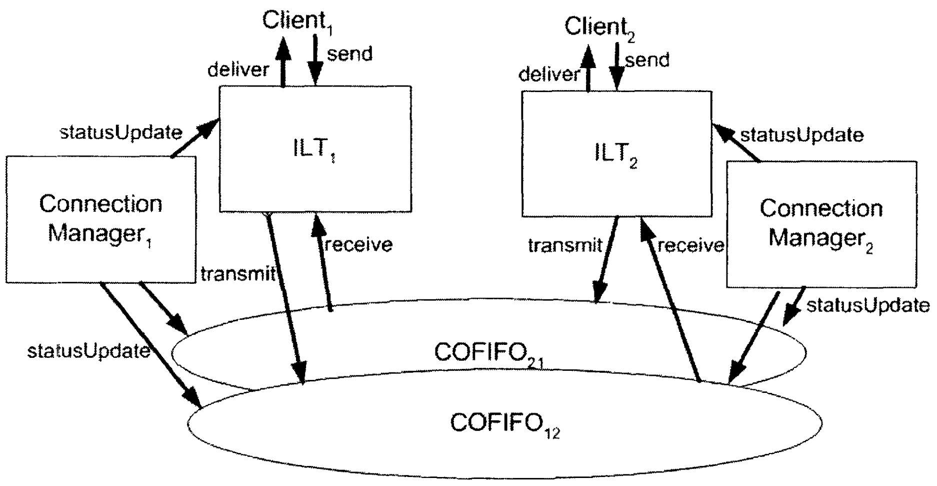

Figure 4-3: A model of the ILT automaton with all of the transitions between different component automata displayed.

4.4 Intermediate Intermittent Global Order Broadcast Specification

4.4.1 Overview

We will encounter many of the same issues in proving that ILT implements IGOB that we saw in proving that LT implements GOB. The actual proof and abstraction mappings will be very similar. In order to reuse as much of the previous proof as possible, we need to specify another intermediate IOA. This will be identical to the previous GBI with modifications to the initial state.

4.4.2 IGBI Automaton

Automaton GlobalBroadcastIntSpec[Ω, M, SendQ$_{p-init}$, next$_{p-init}$, GlobalQ$_{init}$, clock$_{p-init}$], where Ω is a set of processes, M is a set message alphabet, SendQ$_{p-init}$ is a M x T queue, next$_{p-init}$ is a positive integer, GlobalQ$_{init}$ is a M queue, and clock$_{p-init}$ is a logical time.

```
define getMessage( <m, t> ) as m
define getLogicalTime( <m, t> ) as t
Let T be defined as in the ILT$_p$ IOA.

**Signature:**

Input: send$_p$(m), process p $\in \Omega$, message m $\in$ M
Output: deliver$_p$(m), process p $\in \Omega$, message m $\in$ M
Internal: order(p), process p $\in \Omega$
    ticker(p), process p $\in \Omega$

**States:**
$\forall p \in \Omega$, M x T queue, SendQ$_p$, initially SendQ$_{p-init}$
$\forall p \in \Omega$, integer next$_p$, intially next$_{p-init}$
GlobalQ, a M queue, initially GlobalQ$_{init}$
$\forall p \in \Omega$, clock$_p$ $\in$ T, initially clock$_{p-init}$

**Transitions:**
*input send$_p$(m)*
effects: clock$_p$ := clock$_p$ + 1
    append<m, clock$_p$ > to SendQ$_p$

*internal order(p)*
preconditions: <m, t> = first(SendQ$_p$)
    $\forall q \in \Omega$ - {p}, first(SendQ$_q$) $\neq$ null $\Rightarrow$ getLogicalTime(first(SendQ$_p$)) <
        getLogicalTime(first(SendQ$_q$))
    $\forall q \in \Omega$ clock$_q$.time $\geq$ getLogicalTime(first(SendQ$_p$)).time
effects: append getMessage(first(SendQ$_p$)) to GlobalQ
remove first(SendQ$_p$)

*output deliver$_p$(m)*
preconditions: m = GlobalQ[next$_p$]
effects: next$_p$ := next$_p$ + 1

*internal ticker(p)*
preconditions: none
effects: clock$_p$:= clock$_p$ + 1

**Tasks:**
$\forall p \in \Omega$, {order(p)},{ deliver$_p$(m), m $\in$ M}, {ticker(p)}
```

4.5 Proof Overview

The proof that ILT implements IGOB will be very similar to the proof that LT implements GOB. This makes sense, because the ILT algorithm is modeled after the LT algorithm and the IGOB specification is virtually identical to the GOB specification.

The proof that the IGOB intermediate algorithm (IGBI) implements IGOB is the same as the proof that GBI implements GOB. The only difference is outlining how the initial states correspond to each other.

The proof that ILT implements IGBI is slightly trickier. Note that ILT is only guaranteed to implement IGBI after some time has passed during a period of stability. Therefore, the simulation proof does not begin with the start of execution. Furthermore, several of the invariants need to be modified. Lemma 3.5.1(no two non-heartbeat messages have the same logical time) is still true because ILT increments internal logical clocks each time a message is sent. Invariants 3.5.1, 3.5.2, 3.5.3 and 3.5.4 are all results of messages being received in the order that they were sent. Messages may be lost (which is permissible in the COFIFO algorithm), but as long as the messages that are eventually received by a process are received in order, the invariants are still true. The difficult invariants will be 3.5.5 (the Unordered(p, q) sets are the same independent of the choice of q) and 3.5.6 (the GlobalQ's are all prefixes of each other). These are harder because they are simply not true at the beginning of the simulation. Instead, we will have to consider modified invariants that consider only the 'new' messages. Recall that a 'new' message is any message sent after stability has been reached.

The mapping from states of ILT to states of IGBI will also need a few changes. IGBI.SendQ$_p$ will consist of only the 'new' messages in ILT.Unordered(p, q) and IGBI.GlobalQ will be the one of the longest ILT.GlobalQ$_p$'s with all of the 'old' messages removed.

Other than these changes, the simulation proof should be fairly straightforward. The abstraction function (which relates transitions in ILT to transitions in IGBI) remains unaffected.

4.6 Proof that IGBI Spec implements IGOB Spec

First, let us define the projection SendQ$|_m$. If SendQ is a M x T queue, then SendQ$|_m$ is a queue that contains only the messages in SendQ.

Lemma 4.6.1 *g is an abstraction function from automaton IGBI to automaton IGOB*

Proof 4.6.1 Let us define how g maps states in IGBI to states in IGOB.

g(s $\in$ IGBI) = r $\in$ IGOB such that:

IGOB	IGBI
$\text{SendQ}_p()$	$= \text{SendQ}_p\|_m$
next_p	$= \text{next}_p$
GlobalQ	= GlobalQ

In IGOB, we can set the initial states (SendQ_{p-init}, next_{p-init} and GlobalQ_{init}) to be identical to the initial states of IGBI. So we need only to show that for every transition in IGBI, there is a corresponding sequence of actions in IGOB that preserves the mapping and creates the same trace.

Formally, if we are in a reachable state s of IGBI and r is the corresponding state of IGOB created by the above mapping, then for every action π in IGBI, we can find a sequence of actions β in IGOB such that the poststate (s') of π in IGBI maps to the poststate of β (r') in IGOB, or g(s')=r'. Furthermore, π and β must create the same trace and the sequence of actions β must be enabled whenever π is enabled.

Now, let us say that we are in a reachable state s of IGBI and g(s) = r.

• π = IGBI.send_p(m).

The corresponding β in IGOB is IGOB.send_p(m). Both transitions create the same trace, and neither have any preconditions. We need only check that g(s') = r'.

s'.SendQ_p is s.SendQ_p + <m, clock_p >
r'.SendQ_p is r.SendQ_p+ m.

s'.clock_p = s.clock_p + 1.

g(s').SendQ_p = g(s.$\text{SendQ}_p|_m$ + m) = r.SendQ_p + m = r'.SendQ_p.

All other aspects of the state are unaffected by this transition. Since the value of the clock_p variable does not affect the mapping g(), g(s') is the same as r'.

• π = IGBI.order(p)

The corresponding β in IGOB is IGOB.order(p). Neither action affects the trace. The preconditions of IGOB.order(p) are strictly contained in the preconditions of IGBI.order(p), so IGOB.order(p) is enabled whenever IGBI.order(p) is enabled. Now we need to check that g(s') is the same as r'.

s'.SendQ$_p$ = s.SendQ$_p$ - first entry.

r'.SendQ$_p$ = r.SendQ$_p$ - first entry.

s'.GlobalQ = s.GlobalQ + m, where m= getMessage(first(s.SendQ$_p$)).

r'.GlobalQ is r.GlobalQ + m, where m=first(r.SendQ$_p$).

g(s').SendQ$_p$ = g(s.SendQ(p)$|_m$ - first entry) = r.SendQ$_p$ - first entry = r'.SendQ$_p$.

g(s').GlobalQ = g(s.GlobalQ + getMessage(first(s.SendQ$_p$)) = r.GlobalQ + first(r.GlobalQ) = r'.GlobalQ.

All other state variables are unaffected by this transition. Hence, g(s') = r'.

- π = IGBI.deliver$_p$(m)

The corresponding β in IGOB is IGOB.deliver$_p$(m). Both actions create the exact same trace. IGBI.deliver$_p$(m) is enabled when m=s.GlobalQ[s.next$_p$]. Since r.GlobalQ = s.GlobalQ, s.next$_p$ = r.next$_p$ and IGOB.deliver$_p$(m) is enabled when m=r.GlobalQ[r.next$_p$], IGOB.deliver$_p$(m) is enabled exactly when IGBI.deliver$_p$(m) is enabled. Now we need to check that g(s') = r'.

s'.next$_p$ = s.next$_p$ + 1.

r'.next$_p$ = r.next$_p$+ 1.

g(s') = g(s.next$_p$+ 1) = r.next$_p$ + 1 = r'.next$_p$.

All other state variables are unchanged by this action. Hence, g(s') = r'.

- π = IGBI.ticker(p)

The corresponding β in IGOB is simply the empty action, λ. Neither of these actions have any effect on the trace, and λ is enabled whenever IGBI.ticker(p) is enabled. We need only show that g(s') = r'

s'.clock$_p$ = s.clock$_p$+ 1.

g(s') = g(s) = r = r' since g( ) is unaffected by the clock_p variable.

All other state variables are unchanged. Hence, g(s') = r'

Therefore, since all transitions preserve the mapping, g is an abstraction function from IGBI to GOB. ■

Corollary 4.6.1 *IGBI implements IGOB in the sense of trace inclusion.*

Proof 4.6.1 From lemma 4.6.1 we know that there is an abstraction function from IGBI to IGOB. Therefore, from theorem 2.0.1 we know that IGBI implements IGOB.

4.7 Proof that ILT Spec implements IGBI Spec

First, we must formally define the automaton ILT. Let ILT be the composition of ILT_p and COFIFO automata, $\text{ILT} = \prod_{i \in \Omega} \text{ILT}_i[\Omega, \text{M}] \otimes \prod_{j \in \{\Omega - i\}} \text{COFIFO}_{ij}[\text{A}]$, where $\text{A} = \text{M} \cup$ 'heartbeat. Also, let us treat all trasmit_{pq} and receive_{qp} actions as internal actions and hide them from the external trace.

For the simulation proof, we will ignore all 'late markings added to any message. Later, we will provide a proof that the simulation is valid with a small modification when we consider the 'late markings.

Let us define a new M x T set ILT.Unordered(p, q), where p and q are processes in Ω. Informally, this will be a set of all "new" messages sent by p to q that have not been ordered by any process in Ω. Formally, we construct ILT.Unordered(p, q) by appending two queues and a set together and then filtering out irrelevant entries.

For this composition, let us add a history variable to GlobalQ_p. Let us assume that GlobalQ_p contains <m, t> pairs instead of just messages.

Definition 4.7.1 *new_p = $clock_p$ when stability begins.*

Definition 4.7.2 *A message pair <m, t> is "new" if t>$new_{t.process}$*

Definition 4.7.3 *ILT.Unordered(p, q) = ILT.pending$Q_p(q)$ $\cup$ fifoQ_{pq} $\cup$ ILT.received$_q$ - {<m, t> such that m='heartbeat} - {<m, t> such that t.process$\neq$q} - {<m, t> such that <m, t> $\in$ GlobalQ_r for any process r $\in \Omega$} - {<m, t> such that t.time$\leq$new$_p$}*

Definition 4.7.4 *GlobalQ_p $|_n$ = all of the "new" messages in GlobalQ_p with any 'late characters removed.*

4.7.1 Lemma and Invariants

Now we will prove a lemma and several invariants that will be necessary during the proof. Note that the lemma and invariants 1 through 5 are true in all reachable states of ILT, and invariants 6 and 7 are true during periods of stability.

Lemma 4.7.1 *There are no non-heartbeat messages in either IGBI or ILT with the same logical time. Or, more formally, for all <m, t> where $m \in M$ that are either sent, transmitted, received, delivered or ordered by any process $p \in \Omega$, there is no other <m', t'> with $m \in M$ that is sent, transmitted, received, delivered or ordered by any process in Ω where t=t'.*

Proof 4.7.1 In order to prove this, consider extensions of both our current Automata with modified send_p(m) transitions. Say that each Automaton has some set MyMessages_p that contains all the pairs of messages in M and the logical times at which they were created. Modify send_p(m) so that it adds <m, clock_p > to MyMessages_p as well as SendQ_p or pendingQ_p(q). Now, note that within each set of MyMessages_p, no two messages have the same logical time. This is true because clock_p is never decreased by any transition, but it is increased during each send_p(m) transition before clock_p is paired with the message and added to MyMessages_p. It is also true that logical times cannot be the same for messages created by different processes. This is true because logical time is a pair that includes the process ID, which is unique to each process. So a logical time created by one process can never be the same as a logical time created by another process.

Now realize that any message sent, received, transmitted, delivered, or ordered by any process must be contained in MyMessages_p for some process p in Ω. No message can enter the system unless it is sent by a process, at which point it becomes a member of the MyMessage set of that process. Thus, all the messages encountered in both ILT and IGBI have unique logical times.

Invariant 4.7.1 $\forall$ *<m, t>* $\in$ *pendingQ_p(q)* $\cup$ *fifoQ_{pq}, t$\leq$ clock_p*

Proof 4.7.1 Initially, both queues are empty and clock_p = <0, p>, so the Invariant is satisfied.

ILT.send_p(m) increments clock_p and appends <m, t> to pendingQ_p(q) where t=clock_p. Since t$\leq$$\text{clock}_p$ this preserves the invariant.

ILT.transmit_{pq}(<m, t>) moves <m, t> from pendingQ_p(q) to fifoQ_{pq}. This also preserves the invariant.

ILT.receive_{pq}(<m, t>) removes <m, t> from fifoQ_{pq} and increases clock_p, which preserves the invariant.

ILT.heartbeat_p() adds <'heartbeat, clock_p> to pendingQ_p(q). Since $\text{clock}_p \leq \text{clock}_p$, this preserves the invariant.

None of the other transitions affect the relevant state variables. Therefore, since all transitions preserve the invariant, the invariant is true in every reachable state.

Invariant 4.7.2 *In every reachable state of ILT, ILT.clock$_p$ $\geq$ ILT.lt$_q$(p) for any pair of processes q and p in Ω such that $q \neq p$*

Proof 4.7.2 Initial state: both ILT.clock$_p$ and ILT.lt$_q$(p) = 0. Invariant holds.

Transitions: ILT.send$_p$(m) increments clock$_p$ and has no effect on ILT.lt$_q$(p). So if clock$_p$ $\geq$ ILT.lt$_q$(p) before the transition, then it is also true after the transition.

ILT.receive$_{qp}$(<m, t>) may increase clock$_p$ and does not change ILT.lt$_q$(p) so if the invariant was true before this transition, then it is also true after the transition.

ILT.receive$_{pq}$(<m, t>) does not change clock$_p$, but it sets ILT.lt$_q$(p) to t. Since t $\leq$ clock$_p$ (from Invariant 1), lt$_q$(p) $\leq$ clock$_p$ after this transition.

No other transitions influence the relevant state variables. Since the invariant holds in the initial state and is preserved by all the transitions, it is true in every reachable state of ILT.

Invariant 4.7.3 *All messages from a process p are received by another process q in the order in which they were sent. Formally, when process q receives <m, t> from p, this implies that q will never receive <m', t'> from p where t' < t.*

Proof 4.7.3 It is sufficient to show that fifoQ$_{pq}$ + pendingQ$_p$(q) is sorted in order of increasing logical times and every time a <m, t> pair enters either fifoQ$_{pq}$ or pendingQ$_p$(q), there will never be another message <m', t'> with t'<t that enters fifoQ$_{pq}$ or pendingQ$_p$(q). This is sufficient because all messages received by process q from process p are taken off the front of fifoQ$_{pq}$.

Initially, both queues are empty, so the invariant holds.

The only time any <m, t> pair can ever enter pendingQ$_p$(q) is during a heartbeat() or a send$_p$(m) action. Both of these actions pair a message with the current clock$_p$ before adding it to the end of the pendingQ$_p$(q). Since clock$_p$ is a strictly non-decreasing function, it follows that pendingQ$_p$(q) is sorted in order of increasing logical times and after <m, t> enters the pendingQ$_p$(q), no other <m', t'> with t' < t will ever enter pendingQ$_p$(q).

The only transition that may add <m, t> pairs to fifoQ$_{pq}$ is transmit$_{pq}$(<m, t>). Since this takes messages off the front of pendingQ$_p$(q) and appends them to the end of fifoQ$_{pq}$, we can say that fifoQ$_{pq}$ is also sorted in order of increasing ligocal times. Furthermore, after <m, t> enters the fifoQ$_{pq}$, no other <m', t'> with t' < t will ever enter fifoQ$_{pq}$.

The lose$_{pq}$ may remove messages from fifoQ$_{pq}$, but note that this does not affect the ordering of the other messages. Therefore, the invariant is true in all reachable states of ILT.

Invariant 4.7.4 *ILT.$lt_p(q)$ is always less than or equal to the time stamp of any messages from q that p has not yet received. Or more formally, if <m, t> $\in$ ILT.pending$Q_q(p) \cup$ fifoQ_{qp}, then $t \geq$ ILT.$lt_p(q)$.*

Proof 4.7.4 Initially ILT.lt_p(q) is <0, q>, ILT.clock_q is <0, q>, ILT.PendingQ_q(p) is empty, and ILT.fifoQ_{qp} is empty, so the invariant is satisfied.

The only transition that ever changes ILT.lt_p(q) is $\mathrm{receive}_{qp}$(<m, t>). At this time ILT.lt_p(q) is changed to t. From invariant 3, we can see that p will never receive a message <m', t'> from q with t' < t. From liveness properties, we know that p must eventually receive all messages in ILT.pendingQ_{qp} and ILT.fifoQ_{qp}. Therefore, there are never any messages in either ILT.pendingQ_q(p) and ILT.fifoQ_{qp} with logical time less than ILT.lt_p(q).

Invariant 4.7.5 *In every reachable state of ILT, $\forall$ <m, t>$\in$ $received_p$, t$\leq$$clock_{t.process}$.*

Proof 4.7.5 Initially, $\mathrm{received}_p$ is empty, so the invariant is true.

There are only two transitions that may add an element to $\mathrm{received}_p$, ILT.$\mathrm{receive}_{qp}$(<m, t>) and ILT.send_p(m). ILT.$\mathrm{receive}_{qp}$(<m, t>) removes <m, t> from fifoQ_{qp} and places it in $\mathrm{received}_p$. From invariant 1, we know that t.time$\leq$$\mathrm{clock}_q$, so this preserves the invariant. ILT.send_p(m) places <m, clock_p> in $\mathrm{received}_p$. Since clock_p $\leq$$\mathrm{clock}_p$, this also preserves the invariant. Hence, the invariant is true in all reachable states of ILT.

Invariant 4.7.6 *Unordered(p, q) = Unordered(p, r) for any three processes (not necessarily distinct) p, q, r in Ω in every reachable state of ILT during a period of stability.*

Proof 4.7.6 At the beginning of stability, clock_p = new_p for all p in Ω. From invariants 1 and 5, we know that for every message <m, t> in pendingQ_{pq} $\cup$ fifoQ_{pq} $\cup$ $\mathrm{received}_q$, t$\leq$ clock_p. Therefore, if we look at the definition of Unordered(p, q), we can see that Unordered(P, q) is empty at the beginning of stability for all p, q in Ω. So the invariant holds.

Now we need to show that every possible transition preserves the invariant.

LT.send_p(m) appends the same message pair <m, clock_p> to the pendingQ_p(k) of every process k $\neq$ p in Ω, and it also adds <m, clock_p> to its $\mathrm{received}_p$ set. So now <m, clock_p> has been added to Unordered(p, q) for all q $\in$ Ω

ILT.$\mathrm{transmit}_{pq}$(<m, t>) removes <m, t> from ILT.pendingQ_p(q) and adds it to ILT.fifoQ_{pq}. If <m. t> was in Unordered(p, q), this does not remove it. If <m, t> was not in Unordered(p, q) then it is not added.

ILT.order$_p$(<m, t>) If p is the first process to add <m, t> to its GlobalQ then it removes <m, t> from Unordered(q, p) for all q and p in Ω. Otherwise, some other process has already placed <m, t> in its GlobalQ and it has already been removed from all Unordered(q, p), so this transaction has no effect.

ILT.receive$_{qp}$(<m, t>) simply removes <m, t> from ILT.fifoQ$_{qp}$ and adds it to ILT.received$_q$. If <m, t> was in Unordered(p, q), this does not remove it. If <m, t> was not in Unordered(p, q) then it is not added.

ILT.heartbeat$_p$() does not affect Unordered(p, q) for any q in Ω because the definition of Unordered(p, q) does not allow it to include any heartbeat messages.

ILT.lose$_{pq}$ is never enabled because quality$_{pq}$ = 'connected' during stability.

No other transitions affect any of the state variables involved in the definition of Unordered(p, q).

Hence, during a period of stability, in every reachable state of ILT, Unordered(p, q) and Unordered (p, r) are the same for all p, q, r $\in \Omega$

Invariant 4.7.7 *All of the ILT.GlobalQ$_p$ $|_n$ are prefixes of each other. Formally, for all ILT.GlobalQ$_p$ $|_n$ and ILT.GlobalQ$_q$ $|_n$, either ILT.GlobalQ$_p$ $|_n$ is a prefix of ILT.GlobalQ$_q$ or ILT.GlobalQ$_q$ is a prefix of ILT.GlobalQ$_p$ $|_n$.*

Proof 4.7.7 For this proof, let us consider an extension of ILT where the GlobalQ contains message and logical time pairs rather than just messages. Say that order$_p$ <m, t> appends <m, t> to GlobalQ$_p$ instead of just m. I will first prove that each ILT.GlobalQ contains M x T pairs sorted by increasing values of t.

Proof that ILT.GlobalQ$_p$ $|_n$ is sorted by increasing logical time values: ILT.GlobalQ$_p$ $|_n$ is altered only by the order$_p$(<m, t>) transition when <m, t> is a new message. At this time, we can assume that goodConn$_p$ = Ω - p. order$_p$(<m, t>) will only append <m, t> to GlobalQ$_p$ $|_n$ if t $\leq$ t' for all <m', t'> in received$_p$ and if t.time $\leq$ lt$_p$(q) for all q in Ω-{p}. From invariant 3 we know that we can never receive any messages from q with logical time less than or equal to lt$_p$(q). So if t.time $\leq$ lt$_p$(q) for all q(since all q are in goodConn$_p$), then we will never receive any messages <m', t'> with t' < t. If <m, t> has the smallest t for all <m', t'> in received and we can never receive any message with a smaller logical time, then we can conclude that ILT.GlobalQ$_p$ $|_n$ is sorted by increasing values of t.

Also, note that we cannot lose any 'new' messages because lose$_{qp}$ is never enabled during stability. Therefore, by liveness, if stability lasts long enough we will eventually receive all 'new' messages.This implies that there are no missing messages in any of the GlobalQ$_p$ $|_n$'s. That is, if some GlobalQ$_p$ $|_n$ contains 'new' message <m, t>, then it must also contain all 'new' messages <m', t'> with t' < t.

Therefore, since all of the GlobalQ$_p$ $|_n$ have all the messages in the same order, and none can be missing any messages, all of the GlobalQ$_p$ $|_n$'s must be prefixes of each other.

4.7.2 Mapping and Simulation

Lemma 4.7.2 *f is an abstraction function from automaton ILT to automaton IGBI*

Proof 4.7.2 First let's define the mapping f() of how states in IntermittentGlobalBroadcastIntermediateSpec correspond to states in ILT.

f(s $\in$ ILT) = r $\in$ IGBI such that:

IGBI	ILT
SendQ$_p$	= Unordered(p, q) sorted by increasing logical times
next$_p$	= next$_p$
GlobalQ	= one of the longest GlobalQ$_p$ $|_n$ of all processes p in Ω
clock$_p$	= clock$_p$

For the purposes of the simulation proof, we need some sort of first state from which the simulation begins. In this case, the initial state of ILT will be the post-state of the ILT.deliver$_p$(m) that delivers the last "old" message. From here we simply let IGBI.SendQ$_{p-init}$ = ILT.Unordered(p, q) sorted by increasing logical times, IGBI.next$_{p-init}$ = ILT.next$_p$, IGBI.GlobalQ$_{init}$, and IGBI.clock$_{p-init}$ = ILT.clock$_p$. Thus, the initial state preserves the mapping. Now we need only to show that for every transition in ILT there is a corresponding sequence of transitions in IGBI that preserves the mapping and creates the same trace.

Formally, if we are in a reachable state s of ILT and r is the corresponding state of IGBI created by the above mapping, then for every action π in ILT, we can find a sequence of action β in IGBI such that the poststate (s') of π in ILT maps to the poststate of β (r') in IGBI, or f(s')=r'. Furthermore, π and β must create the same trace and the sequence of actions β must be enabled whenever π is enabled.

Now let us say that we are in a reachable state s of ILT and f(s)=r

- π = ILT.send$_p$(m)

The corresponding β is IGBI.send$_p$(m). Both transitions create the same trace, and both are always enabled. We need only check that f(s')=r'.

s'.clock$_p$ = s.clock$_p$+ 1.
r'.clock$_p$ = r.clock$_p$+ 1.

For all q $\in \Omega$ such that q $\neq$ p, s'.pendingQ$_p$(q) = s.pendingQ$_p$(q) + <m, s.clock$_p$ + 1 >. (Note that for all q $\in \Omega$ such that p $\neq$ q, s'.Unordered(p, q) = s.Unordered(p, q) $\cup$ {<m, s.clock$_p$ >+ 1})

s'.received$_p$ = s.received$_p$ $\cup$ {<m, s.clock$_p$ + 1 >}. (Note that s'.Undelievered(p, p) = s.Unordered(p, p) $\cup$ {<m, s.clock$_p$+ 1>})

s'.Unordered(p, q) = s.Unordered(p, q) $\cup$ {<m, s.clock$_p$ + 1 >} for all q $\in \Omega$).
r'.sendQ$_p$ = r.sendQ$_p$ + <m, r.clock$_p$ + 1 >.

f(s'.clock$_p$) = f(s.clock$_p$+ 1) = r.clock$_p$ + 1 = r.clock$_p$

f(s'.Unordered(p, q)) = f(s.Unordered(p, q) $\cup$ {<m, s.clock$_p$+1 >}) = r.sendQ$_p$ + <m, r.clock$_p$+ 1 > = r'.sendQ$_p$.

f(s.Unordered(p,q)) is defined as a set with no ordering, and r.sendQ$_p$ does have order. Because IGBI.send$_p$(m) appends <m, r.clock$_p$ + 1 > to the end of r.SendQ$_p$, we must confirm that <m, r.clock$_p$ + 1 > has the largest logical time of all elements of s'.Undelivered(p, q). This is true because the clock$_p$ is a strictly nondecreasing variable and <m, s.clock$_p$+ 1> is the most recent addition to Unordered(p, q). Therefore <m, s.clock$_p$ + 1 > must have the largest logical time of all M x T pairs in Unordered(p, q).

All other state variables remain unchanged. Hence f(s') = r'.

- π = ILT.transmit$_{pq}$(<m, t>)

The corresponding β is simply the empty transition λ. Since λ is always enabled, we know that β is enabled whenever π is enabled. Since we will hide all transmit and receive actions in the final trace, neither action here will impact the final trace. Now we need only show that f(s') = r'.

s'.pendingQ$_p$(q) = s.pendingQ$_p$(q) - <m, t>

s'.fifoQ_{pq} = s.fifoQ_{pq} + <m, t>.

From the definition, we can see that this change does not affect Unordered(p, q). Therefore f(s'.Unordered(p, q)) = f(s.Unordered(p, q)) = r.SendQ_p = r'.SendQ_p. Since all other state variables are unaffected by this transition, we can conclude that f(s') = r'.

• π = ILT.deliver$_p$(m)

The corresponding β is IGBI.deliver$_p$(m). (Note: since the ILT.GlobalQ_p $|_n$ are all prefixes of each other, and IGBI.GlobalQ = one of the longest ILT.GlobalQ_p $|_n$ of all p $\in \Omega$, it is safe to say that ILT.GlobalQ_p $|_n$ is a prefix of IGBI.GlobalQ for any p $\in \Omega$) Since ILT.GlobalQ_p $|_n$ is a prefix of IGBI.GlobalQ, if m = ILT.GlobalQ_p $|_n$[s.next$_p$] then m = IGBI.GlobalQ[r.next$_p$]. Therefore, IGBI.deliver$_p$(m) is enabled whenever ILT.deliver$_p$(m) is enabled. They both create the same external trace, so all we need to check now is that f(s') = r'.

s'.next$_p$ = s.next$_p$+ 1

r'.next$_p$ = r.next$_p$+ 1

f(s'.next$_p$) = f(s.next$_p$+ 1) = r.next$_p$+ 1 = r'.next$_p$.

All other state variables are not affected by this transition. Hence, f(s') = r'.

• π = ILT.order$_p$(m), and let <m, t> be the message-logical time pair that satisfy the first precondition.

Note that since m has not yet been delivered, and the simulation begins after all 'old' messages have been delivered, m must be a 'new' message. The corresponding β can be one of two actions:

1. if $\forall$ q $\in \Omega$, m$\notin$ GlobalQ_q $|_n$, then β = IGBI.order(t.process).

2. Otherwise some other process has already performed the order operation on this message, so β is simply the empty action, λ.

I will start with case 2, since it is the simpler one to prove. Since λ is always enabled, we know that β is enabled whenever ILT.order$_p$(m) is enabled. And since ILT.order$_p$ is an internal action, the trace is the same in both cases. Now all that remains is to prove that f(s') = r'

s'.received = s.received - <m, t>.

Note that since some other process q has already performed ILT.order$_q$(m), we know from the definition of Unordered that <m, t> is not in Unordered(t.process, r) for any process r $\in \Omega$, so Unordered(t.process, q) is unchanged for all q $\in \Omega$. Hence, the mapping to IGBI.Send$Q_{t.process}$ is unchanged.

s'.GlobalQ$_p$ $|_n$ = s.GlobalQ$_p$ $|_n$ + <m, t>. (Note that it doesn't matter whether or not m is marked late since any 'late marking is removed by the $|_n$ mapping.)

ILT.marker$_p$ may or may not change. Since it doesn't affect the mapping, it doesn't matter.

Note that some other process q has already performed order$_q$(m), so s.GlobalQ$_q$ $|_n$ already included <m, t>. Since all GlobalQ$|_n$'s are prefixes of each other, it follows that s.GlobalQ$_q$ $|_n$ is strictly larger than s.GlobalQ$_p$ $|_n$. Therefore, when we add a single message to s.GlobalQ$_p$ $|_n$, s'.GlobalQ$_p$ $|_n$ is at most as long as s.GlobalQ$_q$ $|_n$. Therefore, s'.GlobalQ$_p$ $|_n$ is not longer than one of the longest GlobalQ's in s. From invariant 4.7.7, we know that s'.GlobalQ$_p$ $|_n$ is a prefix of one of the longest GlobalQ$|_n$'s in s. Hence, the mapping to IGBI.GlobalQ remains unchanged. Since λ also does not affect the mapping, it follows that f(s') = r'.

In the first case, we know that p is the first process to perform ILT.order$_p$ on message <m, t> and the corresponding β is IGBI.order(t.process). Since both are internal actions, they do not affect the external trace. We need to show that IGBI.order(t.process) is enabled whenever ILT.order$_p$ <m, t> is enabled.

The first precondition for IGBI.order(t.process) is that <m, t> = first(SendQ$_{t.process}$). From the definition of Unordered(p, q) we can see that in state s, <m, t> is in Unordered(t.process, p) for all p $\in \Omega$. We need only show that for all <m', t'> in Unordered(t.process, p), t < t'. From invariant 4.7.3, we know that if t.process sent any earlier messages, they must have been received by p. Hence, any <m', t'> with t' < t must be in either ILT.received$_p$ or ILT.GlobalQ$_p$ $|_n$. If it is in ILT.GlobalQ$_p$ $|_n$, then we know it cannot be in Unordered(t.process, p). If it is in ILT.received$_p$, then received$_p$ has a message with a time stamp less than t, which contradicts the third precondition of IGBI.orcer(t.process). Hence, <m, t>=r.first(SendQ$_{t.process}$) whenever ILT.order$_p$ <m, t> is enabled.

With invariant 4.7.2 (ILT.lt$_q$(p) $\leq$ ILT.clock$_p$), it follows that the third precondition of IGBI.order(t.process) is satisfied whenever the second precondition of ILT.order$_p$(m) is satisfied.

From invariant 4.7.4 and the definition of Unordered(p, q), we can see that either r.first(SendQ$_q$) is in s.received$_p$ or s.lt$_p$(q) < t. If s.lt$_p$(q) < t then π is not enabled. If r.first(SendQ$_q$) is in s.received$_p$ then the third precondition of ILT.order$_p$ <m, t> implies that the second precondition of IGBI.order(t.process) is satisfied.

Hence, β is enabled whenever π is enabled.

Now we must show that f(s') = r'.

s'.received$_p$ = s.received$_p$ - <m, t>.

s'.GlobalQ$_p$ $|_n$ = s.GlobalQ$_p$ $|_n$ + m. (It doesn't matter whether or not m is marked late because the marking is removed with the $|_n$ mapping)

s'.Unordered(t.process, q) = s.Unordered(t.process, q) - <m, t>.

r'.SendQ$_{t.process}$ = r.SendQ$_{t.process}$ - <m, t>. r'.GlobalQ = r.GlobalQ + m.

ILT.marker$_p$ may or may not be increased. It doesn't matter since that does not affect the mapping.

So f(s'.Unordered(t.process, q)) = r'.SendQ$_{t.process}$ for any q $\in \Omega$. Because p is the first process to append m to its GlobalQ$|_n$, and all of the GlobalQ$|_n$'s are prefixes of each other, we can conclude that s.GlobalQ$_p$ $|_n$ is not shorter than s.GlobalQ$_q$ $|_n$ for all q $\in \Omega$ where q $\neq$ p, and after appending m, GlobalQ$_p$ $|_n$ is the longest GlobalQ$|_n$. Therefore, s.GlobalQ$_p$ $|_n$ = r.GlobalQ and s'.GlobalQ$_p$ $|_n$ = r'.GlobalQ$_p$ $|_n$. Since no other state variables change, we can conclude that f(s') = r'

- π = ILT.receive$_{qp}$(<m, t>)

The corresponding β is simply ticker(p) performed (max[t.time - s.clock$_p$, 0] + 1) times. For convenience, we will simply refer to this number as n in the next few paragraphs. Since IGBI.ticker(p) is always enabled, we know that β is enabled whenever π is enabled. Since we will hide all transmit and receive actions in the final trace and IGBI.ticker(p) is an internal action, neither will impact the final trace. Now we need only show that f(s') = r'.

s'.fifoQ$_{pq}$ = s.fifoQ$_{pq}$ - <m, t>

s'.received$_q$ = s.received$_q$ + <m, t>

From the definition of Unordered(p, q), we can see that this change does not affect Unordered(p, q). So f(s'.Unordered(p, q)) = f(s.Unordered(p, q)) = r.SendQ$_p$ = r'.SendQ$_p$.

s'.clock$_p$ = s.clock$_p$ + n

r'.clock$_p$ = r.clock$_p$ + n

f(s'.clock$_p$) = f(s.clock$_p$ + n) = r.clock$_p$ + n = r'.clock$_p$.

s.lt$_p$(t.process) is also changed, but this does not affect any of the states in IGBI. Since all other state variables remain unchanged, we can conclude that f(s') = r'.

- π is ILT.heartbeat$_p$()

The corresponding β in IGBI is λ. λ is always enabled. ILT.heartbeat$_p$() is an internal action, neither affect the external trace. Now we need only prove that f(s') = r'.

s'.pendingQ$_p$(q) = s.pendingQ$_p$(q) + <'heartbeat, s.clock$_p$ > for all q $\neq$ p.

Note that since all 'heartbeat messages are not a part of Unordered(p, q), we can see that Unordered(p, q) is unaffected by this transition, for any p and q in Ω. Therefore, the mapping is not influenced and f(s') = r'.

- π be ILT.statusUpdate$_{pq}$(quality)

The corresponding β is simply the empty action λ. Both are always enabled and neither affect the external trace.

This proof is only valid during a period of stability. This means that all connection qualities must always be good. Therefore, the only valid quality is 'connected'. Performing ILT.statusUpdate$_{pq}$('connected') when quality$_{pq}$ is already connected does not change any part of the state. Therefore, f(s')=f(s)=r=r'.

- π = ILT.lose$_{pq}$

The precondition for this action requires that quality$_{pq}$='disconnected'. Since this proof is for a period of stability, ILT.lose$_{pq}$ is never enabled so we do not have to consider its affects.

Therefore, f is an abstraction function from ILT to IGBI. ■

Corollary 4.7.1 *ILT implements IGBI in the sense of trace inclusion. Furthermore, by transitivity, ILT implements IGOB.*

Proof 4.7.1 From lemma 4.7.2 we know that there is an abstraction function from ILT to IGBI. Therefore, by theorem 2.0.1 we know that ILT implements IGBI. Furthermore, by transitivity and lemma 4.6.1, we know that ILT implements IGOB.

4.7.3 Dealing with late messages

In the simulation proof, we ignored all of the 'late markings. Here I will attempt to show how 'late markings affect the proof.

Because all of the processes have varying logical clock values, it is possible that a message sent by one process before stability begins will have a larger logical clock value than a message sent by another process after stability begins. Then, because the processes try to deliver messages in order of increasing logical times, there may be 'new' messages delivered before some 'old' messages. If for some reason an 'old' message with a late logical time was delivered before stability was reached, then that process will mark all of the 'new' messages with earlier logical times 'late. Furthermore, processes may disagree on which 'new' messages are marked 'late.

To outline this issue clearly, the following diagram illustrates an example involving three processes: A, B and C. In order to simplify the diagram, logical times are represented by a single number, and irrelevant transitions have been omitted. The first wavy line represents the beginning of stability, and it implies that each process updates their connection qualities to 'connected'. The second wavy line represents the state immediately after C delivers m1 (the last 'old' message), where we want to say that global order begins. Note that even after global order is guaranteed, A and B disagree on the 'late-ness of m2.

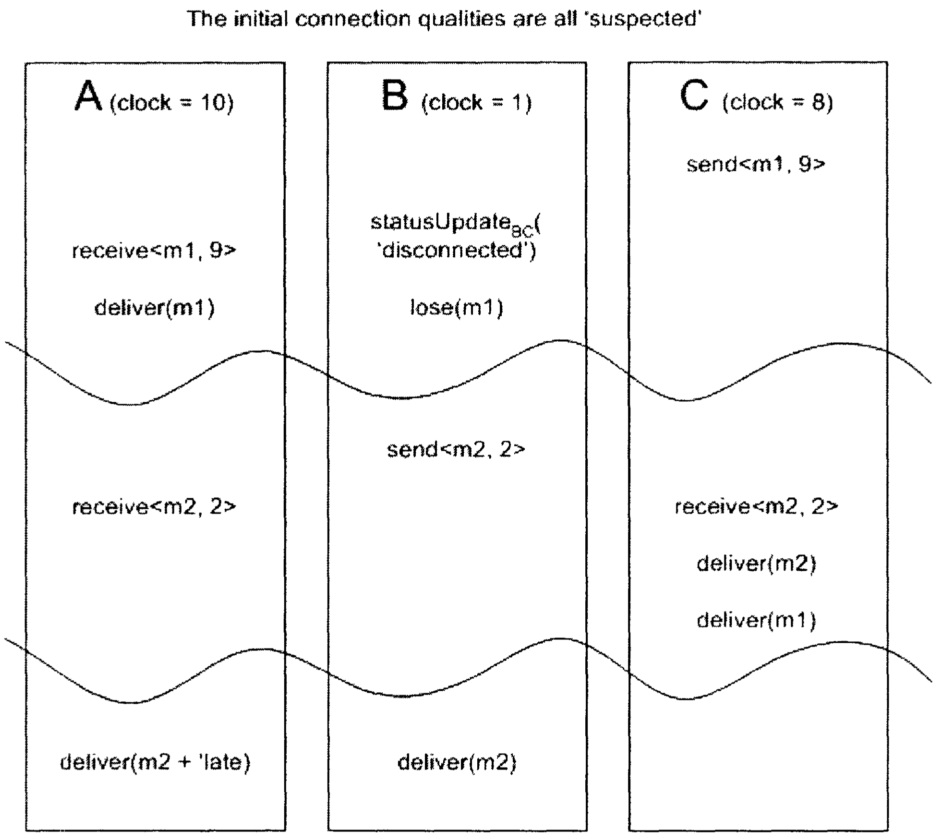

Figure 4-4: Example of 'late discrepancy

The proof presented earlier takes the stance that the important thing is delivering messages in order and 'late-ness is only secondary issue. However, if we treat a 'late marking as a sort of mutation of a message, this is inacceptable. If we require that all processes agree on 'late markings, the ILT algorithm does not implement IGOB until later. If we say that LOM (Latest 'Old' Message) is the latest logical time of all 'old' messages, then ILT does not implement IGOB until all messages with logical times less than or equal to LOM have been delivered. A few simple invariants show that no messages delivered after this point could be marked 'late.

Invariant 4.7.8 *When a process p orders <m, t>, this implies that p will never receive a <m', t'> with t>t' unless t'.process $\notin$ goodConn$_p$*

Proof 4.7.8 This is the same as proving that for all q$\in$goodConn$_p$, p does not have and will never receive a message from q with logical time less than t. We know that p does not have any messages with logical times less than t in received$_p$, or else the third precondition of order<m, t> would not be satisfied. From invariant 4, we can see that if t.time<lt$_p$(q) for all q $\in$ GoodConn$_p$, then we can never receive a message from q with logical time less than t. Hence, the invariant is true in all reachable states of ILT.

Invariant 4.7.9 *During a period of stability, if a message <m, t> is marked 'late, then t < LOM, where LOM is as defined previously.*

Proof 4.7.9 First let's take a step back and look at ILT.order$_p$(m). Note that marker$_p$ is always the value of the latest logical time of messages on the GlobalQ$_p$. In order for a message <m, t> to be marked 'late, there must be a message on GlobalQ$_p$ with a logical time later than t. If <m, t> is marked 'late and <m', t'> is one message on GlobalQ$_p$ with t'>t, then with invariant 8 we know that t.process was not in goodConn$_p$ when m' was ordered. Hence, m' could not have been ordered during the period of stability. Therefore, m' must be an 'old' message. It follows that no message <m, t> can be marked late unless t is less than the logical time of at least one 'old' message. Thus, the invariant is true.

Now that we have proved that ILT implements IGOB when we ignore all 'late messages and 'late messages will eventually end, we can say that ILT implements IGOB even if we consider a message with a 'late marking a different message. IGOB will start at a later point in the trace (namely after there can be no more 'late messages), but the proof is essentially identical.

It's nice to pin down exactly when the algorithm will not have any more 'late messages, but how does a client know when 'late messages stop? Or, if someone were looking at the trace of ILT with no knowledge of the logical times assosciated with each message, how would that person know when the 'late messages have stopped coming? Quite simply, you know that a process will never mark another message as 'late as soon as it has delivered at least one 'new' message from every other process. This is true because once a process has delivered a message from every other process, it must have delivered a message from the process that sent the message with the LOM logical time. Once that has been delivered, we know that there are no more messages with logical times less than LOM. Hence, there are no more 'late messages. However, this analysis assumes that one can see the trace. In actuality, a client at site A knows only how well other processes are connected to site A. It has no knowledge of whether or not site B can communicate with site C. Therefore, any individual client has no idea whether or not a system is stable. Hence, any individual client does not know when global order is guaranteed.

Chapter 5

Connection Manager

Many performance trade-offs are controlled by the connection manager. Parameters in the connection manager determine how long one process will wait for another process to send it a message before it changes the connection quality. This in turn directly affects message latency. The quicker a process switches from connected to suspected, the lower the messages latencies will be. If a process waits longer to switch from connected to suspected, there will be better global consistency with the message ordering. The details of the connection manager also affect other properties and guarantees that would be nice to analyze. For example, one property of the RC project implementation involves what happens when clients at different ends of a TCP channel don't see the same connection qualities. A property like this is a function of the connection manager rather than the ILT algorithm because there is no guarantee that processes A and B will always agree on the connection quality between them. For instance, it is possible for A to receive notification that it was disconnected from B for a short while and for B to think that the connection quality only got as bad as 'suspected'. We will address the ramifications of this after we describe how the connection manager works.

5.1 How it works

The connection manager is a special layer that sits on top of the TCP connection. The first thing that needs to be clarified is how the connection manager interprets the traffic on a TCP channel. The connection manager allows the TCP connection to vary between three different states: TCP_Connected, TCP_Silent, and TCP_Disconnected. Whenever a message is received via a TCP connection, the connection is classified as TCP_Connected. If five seconds pass where no transmissions are received, the connection is classified as TCP_Silent. If an additional thirty seconds pass and no transmission is received, the connection manager kills the TCP channel and switches the status to TCP_Disconnected. If at any time a TCP socket throws an exception, it is closed and the TCP channel is immediately classified as TCP_Disconnected. When two processes want to establish a TCP connection, the connection is initiated by the process with the smaller process ID. The first communication over a new channel will include the sender's identification information. This is designed to prevent multiple TCP processes opening between a pair of processes. The state machine governing internal connection state transitions is outlined in Figure 5-1. Please note that the actual timing parameters may be customized by the client. The numbers given here are just one example.

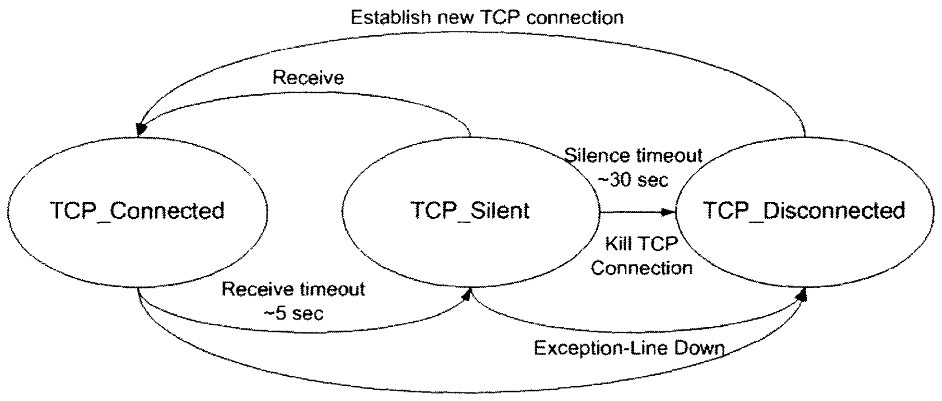

Figure 5-1: Internal Connection States

There is a separate state machine governing external state transitions. The external state represents how the connection manager interprets the different internal states of the TCP connection. As the state of the TCP connection changes, the connection quality must be re-evaluated. Whenever a channel switches to TCP_Connected, the quality is deemed 'connected' and an update is sent to the client. From the 'connected' state, if the channel either becomes TCP_Silent or TCP_Disconnected, the quality is classified as 'suspected' and the client is notified. Once a connection quality is classified as 'suspected', the connection manager starts a clock. If the TCP status does not change to TCP_Connected within sixty seconds, the connection is automatically killed and the quality changes to 'disconnected'. Note after a TCP channel is killed, there is a window of time where a new TCP connection may be established without the client knowing it ever lost communication. If the connection quality is classified as 'disconnected' for more than five seconds, the connection manager deletes all the messages that were enqueued to be sent along that channel. This five second delay is included so that the old messages are not deleted until the process is fairly certain that both processes have agreed on the disconnection. Figure 5-2 illustrates how connection qualities are updated. Once again, the timing parameters may be customized by the user.

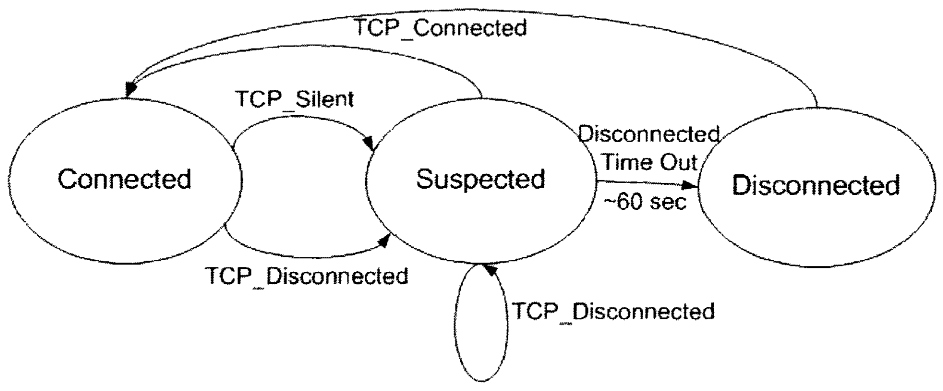

Figure 5-2: External Connection States

By looking at these two diagrams together, we can make several statements about how the connection manager behaves in different situations. The client will be notified every time more than five seconds pass without receiving a communication from a particular process. This time window was chosen because the RC implementation has each process sending out 'heartbeat messages every five seconds. Therefore, if five seconds pass with no communication, the TCP channel is not operating as well as it could. Also note that a TCP channel is killed if no message is received for 35 seconds, but the client will not be notified that the channel is disconnected until 65 seconds have passed without receiving a message.

5.2 Properties

The implementation choices that have just been described influence the properties we can assume in our RC program. The proof presented earlier is meant to describe the behavior of the system during a period of stability. Here I would like to address what may happen during periods of instability.

First, note that because we do not immediately change the quality to 'disconnected' whenever a TCP channel is killed, the processes at the ends of the TCP channel may not agree on whether or not they were ever 'disconnected'. Consider the scenario outlined in Figure 5-3:

In this case, it's clear that the client at point A saw that he had been disconnected from B, but B's client only saw a 'suspected' connection with A. However, because they re-established their TCP connection before A deleted his queue of messages for B, there should be almost no gap in the messages sent from A to B. I say almost, because any message that A sends while he perceives B as 'disconnected' will not be added to the queue of messages from A to B. In this example, B would not receive any message that A might have broadcasted between 1:08 and 1:09. So B could

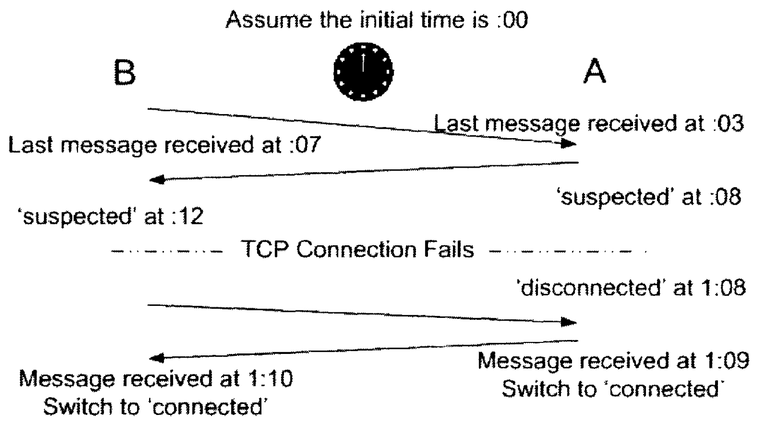

Figure 5-3: Timing Example

be missing a message that A broadcasted to other processes even though B's client never sees a 'disconnected' connection quality. The client at process A, however, will know that B is missing this message because it saw the disconnection. There may also be problems depending on exactly when a channel is initially declared 'connected'. In the RC system, A may try to establish a TCP connection with B. If A is successful, it declares itself 'connected' and then proceeds to send its contact information to B. However, if they are disconnected before B can receive this information, B will never consider itself 'connected' to A.

On the other hand, we can guarantee that A will receive every message sent by B. This is because B never deletes anything from its queue of messages for A unless it sees a 'disconnected' quality. In this respect, this system is sender-oriented, because the sender has a fairly good idea of who receives his messages while the receiver cannot always be sure of whether or not it is missing any messages.

There are methods for helping B know when it may be missing messages from A. One of these methods is to add an additional connection quality to cover the window between when a process considers itself 'disconnected' and when it knows that the process on the other end considers itself 'disconnected'. However, too many connection qualities tend to confuse clients. Another possibility is for processes to send additional messages between each other whenever a new TCP connection is established to inform each other if the connection was classified as 'disconnected'. Then, if A saw a disconnection and B didn't, B could quickly change the status to 'disconnected' and then back to 'connected' to communicate to the client that there may be messages missing. This approach means that B will see a 'disconnected' quality even though it is connected to A, which seems somehow inaccurate. Either of these solutions would solve the problem with disagreeing on a 'disconnect' when one process reconnects after being disconnected between a minute and a minute and five seconds. Unfortunately, neither of these solutions would help with the discrepancies seen when one process declares itself 'connected' and then disconnects before the process on the other end declares itself 'connected'. This situation is actually a special case of the "two generals" commitment problem presented in chapter 5 of [2]. Essentially, it is impossible to create a protocol whereby two processes can commit to being 'connected'.

Chapter 6

Conclusion

It is easiest to model intermittent properties by first considering the non-dynamic case. The process of creating a proof and analysis of this situation goes a long way in understanding the subtleties that arise when disconnections are introduced. As an additional bonus, large parts of the non-dynamic proof may be reused when attempting the proof in a dynamic environment. The next step is to alter the initial conditions of the specification to reflect the state of the system during a period of stability after there may have been an arbitrary time of instability in the past. Keeping the states and transitions in the two specifications identical means that there will be large sections of the proof that can simply be recycled. The similarities also clarifies the relationship between the intermittent specification and the non-dynamic specification. This approach was also used in [4].

The changes in the proofs needed for the dynamic and non-dynamic cases may be divided into two parts: the abstraction function and the invariants. When the specifications for the non-dynamic and intermittent properties are virtually identical, the abstraction function changes either very little or not at all. In the proof for IGOB, the abstraction function was identical to the one used for LT. The invariants, however, are more difficult. Because the invariants are designed to prove the abstraction function, and the abstraction functions are similar, one would hope to be able to prove similar invariants as well. But in this proof, for example, there were two invariants that were simply no longer true when disconnections were introduced. These had to be modified so that they only applied to periods of stability. This enabled us to keep essentially the same invariants during stability, which is what we are concerned with during the course of the proof.

The intermittent global order broadcast property presented here is only one way to solve the problem of ordering messages in a dynamic environment. Like any other system, there are a number of pros and cons. IGOB is a light-weight protocal. It doesn't involve any additional synchronization rounds during network changes, yet it guarantees global ordering properties under well-specified conditions regardless of prior dynamic changes in the connection qualities. Because it is a light-weight protocol, it may allow for asymmetric perceptions of communication at different clients. For example, a client at one site does not know anything about what messages another site might be receiving. So if Alice and Bob are connected with a perfect communication channel, Alice has no idea whether or not Bob is receiving messages from Carol. If Alice knew that Bob was missing messages from Carol, Alice could forward him the messages she had received from Carol. While this is a desirable property, it would require lots of additional communication as well as a more complicated protocol.

Future work may include formal proofs of other intermittent properties. One such possibility is connection-oriented global order. Informally, connection-oriented global order means that any pair of sites that have a stable connection will always view the same messages in the same order. This is a more complicated property to fulfill because it would require an implementation with a more complicated protocol that would involve additional communication between remote sites. The specification would also involve more subtleties. However, it is clear that any system that fulfills the requirements for connection-oriented global order would also implement intermittent global order broadcast. Therefore, the proof presented here could be both an example and a building block for such a proof.

Connection-oriented global order is another way to weaken global order. It would be implemented with a protocol that involves a special synchronization round every time group membership changes. During this synchronization round, no 'new' messages are shared until everyone has received and delivered all of the 'old' messages. Then, global order broadcast begins immediately after the synchronization round is complete. Furthermore, each client knows that global order has started. Once again, this system requires additional rounds of communication. If processes are frequently changing their connection status, consensus rounds are performed frequently. If stability does not last long enough for the consensus round to terminate, the extra communication was wasted anyway. Therefore, this technique is most useful in situations where the group membership does not change frequently.

Bibliography

[1] Xavier Defago, Andre Schiper and Peter Urban. *Totally Ordered Broadcast and Multicast Algorithms: A Comprehensive Survey.* Tech. Rep. DSC/2000/036, Swiss Federal Institute of Technology, Lausanne, Switzerland, September 2000.

[2] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[3] Leslie Lamport. *Time, clocks, and the ordering of events in a distrubted system.* Communcations of the ACM, 21(7):558-565, July 1978.

[4] Roger Khazan and Nancy Lynch. *An Algorithm for an Intermittently Atomic Data Service Based on Group Communication.* International Workshop on Large-Scale Group Communication, October 2003.

[5] Roger Khazan, et. al *Robust Collaborative Multicast project.* MIT Lincoln Laboratory. Information Systems Technology group. Project report in preparation. 2003-2004.

[6] Butler Lampson. *Generalizing Abstraction Functions.* Massachusetts Institute of Technology, Laboratory for Computer Science, principles of computer systems class, handout 8, 2002. `http://web.mit.edu/6.826/archive/S02/8.pdf`