A Theory of Clock Synchronization

by

Boaz Patt

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

October 1994

Author . . . .
Department of Electrical Engineering and Computer Science
October 17, 1994

Certified by . . . .
Nancy A. Lynch
Cecil H. Green Professor Of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
Baruch Awerbuch
Associate Professor of Computer Science, Johns Hopkins University
Thesis Supervisor

Accepted by . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

A Theory of Clock Synchronization

by

Boaz Patt

Submitted to the Department of Electrical Engineering and Computer Science
on October 17, 1994, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We consider the problem of clock synchronization in a system with uncertain message delays and clocks with bounded drift. To analyze this classical problem, we introduce the concept of synchronization graphs, and show that the tightest achievable synchronization at any given execution is characterized by the distances in the synchronization graph for that execution. Synchronization graphs are derived from information which is locally available for computation at the processors (local time of events and system specification), and can therefore be used by distributed algorithms. Using synchronization graphs, we obtain the first optimal on-line distributed algorithms for external clock synchronization, where the task of all processors is to estimate the reading of the local clock of a distinguished processor. The algorithms are optimal for all executions, rather than only for worst cases. The algorithm for systems with arbitrarily drifting clocks has high overhead; we prove that this phenomenon is unavoidable, namely any optimal general algorithm for external synchronization has unbounded space complexity. For systems with drift-free clocks (i.e., clocks that run at the rate of real time), we present a particularly simple and efficient algorithm. We also present results for internal synchronization, where the task of the processors in the system is to generate a synchronized "tick." Our approach is robust in the sense it encompasses various system models, such as point-to-point or broadcast channels, communication links that may lose, duplicate and re-order messages, and crashing processors. In addition, synchronization graphs can be used to detect corrupted information.

Thesis Supervisor: Nancy A. Lynch

Title: Cecil H. Green Professor Of Computer Science and Engineering

Thesis Supervisor: Baruch Awerbuch

Title: Associate Professor of Computer Science, Johns Hopkins University

Acknowledgments

I would like to thank all the people who helped me be where I am today. Professionally, my first mentor was David Peleg from the Weizmann Institute of Science. His knowledge, intellectual integrity, and work methods gave me the first ideas what a computer scientist should be. I will never forget his kind nature and sharp understanding.

In MIT, I was blessed with two other extremely gifted advisors. The thoroughness of Nancy Lynch is second to none. From her I learned the basic methods of rigorous mathematical reasoning about distributed systems. Her pointed observations had clarified much of my confusions.

My second advisor, Baruch Awerbuch, is an explosive source of ideas, endless encouragement and support, always accompanied with an unpredictable sense of humor. His "killer instinct" for distributed algorithms, and his fearlessness of "impossible" tasks will always serve me as an ideal.

Sergio Rajsbaum is responsible for the choice of clock synchronization as my research target. His earlier work inspired the contents of this thesis. His natural curiousity and deep insights (which are the source of many of the ideas in this thesis) typify, in my mind, the classical scientist.

I enjoyed many illuminating discussions with Robert Gallager. His views about distributed systems, scientific research and mathematics are a constant guide for my thinking. I thank him for sharing his wisdom with me.

Lastly, I would like to thank my family. My parents, Avraham and Elisheva Patt, have provided me with all the possible support in some of the hardest times I had. I am deeply grateful for their unconditional love.

Above all, I thank my wife Galia, for her true love and support, which were with me at all times, and my daughter Alma, who gave a new reason to life. I love you!

Contents

Chapter 1

Introduction

1.1 Background

Clock synchronization is one of the most fundamental problems of distributed computing. Roughly speaking, the goal of clock synchronization is to ensure that physically dispersed processors will acquire a common notion of time, using local physical clocks (whose rates may vary), and message exchange over a communication network (with uncertain transmission times). The discrepancy between clock readings is called the *tightness* of synchronization. There are numerous applications for synchronized clocks in computer networks. For example, in database systems, version management and concurrency control usually depend on the ability to consistently assign timestamps to objects. Many distributed applications use timeouts (e.g., communication protocols, resource allocation protocols), and their performance depends to a large extent on the quality of synchronization between remote processors. From the theoretical perspective, having synchronized clocks enables one to use distributed algorithms that proceed in rounds, thus considerably simplifying their design and analysis. For an excellent discussion of the importance of clock synchronization, see Liskov's keynote address at the 9th PODC [18].

The basic difficulty in clock synchronization is that timing information tends to deteriorate over the temporal and spatial axes. More specifically, when the rate of local clocks is not known precisely in advance, the tightness of synchronization loosens as time passes; and when a processor is communicating timing information to remote processors, there is some inherent cumulative timing uncertainty, unless message transmission times are known precisely. Practically, ideal clocks and communication links do not exist. However, there

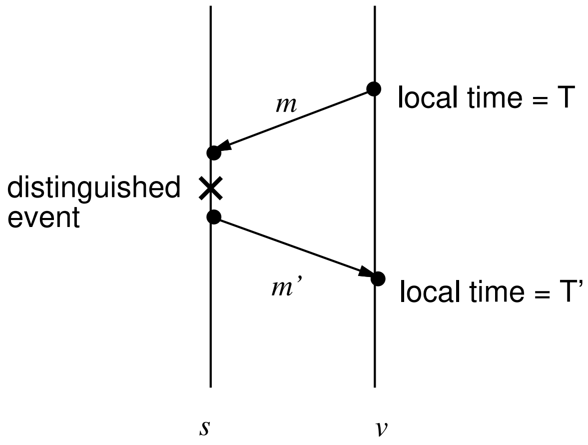

Figure 1-1: *Processor v send a message m to processor s, s sends a message m' back to v. A distinguished event, marked by a cross, occurs at s after m is received and before m' is sent.*

are always some *a priori* guarantees about the timed behavior of the system: usually it is assumed that local clocks have known lower and upper bounds on their rate of progress with respect to real time. We call these bounds *drift bounds*. In addition, it is assumed that there are known lower and upper bounds on the time required to transmit a message. We call these bounds *message latency bounds*. The essence of all clock synchronization problems is how to use these bounds to obtain tight synchronization.

In this thesis we present a theoretical study of clock synchronization problems. Our starting point is an elementary variant of the problem, described informally as follows.

> *Obtain bounds on the reading of the local clock when some distinguished remote event occurs in the execution.*

Example. Consider a system that consists of two processors s and v, connected by a bidirectional communication link. Suppose that processor v sends a message m to s when the local clock at v shows T; processor s then responds by sending a message m' to v, which is received at v when its local clock shows T'. See the time-space diagram in Figure 1-1. (A brief explanation of time-space diagrams is given in Appendix A.) Suppose further that some distinguished event occurs at processor s after m is received and before m' is sent. Clearly, when m' is received, processor v can deduce that the distinguished event occurs within its local time interval $[T, T']$. The difference $(T' - T)$ is the *tightness* of synchronization. ∎

To study synchronization problems, we define a system model, and analyze it at an abstract graph-theoretic level. Using the results we obtain for graphs, we analyze clock synchronization problems that are more practical than the elementary variant above. Specifically, we give results for two kinds of clock synchronization tasks, motivated by the following settings.

External Synchronization: There exists a distinguished processor called *source* in the system. The task for each other processor is to obtain, at each time, the smallest interval $[a, b]$ such that the current reading of the source clock is in $[a, b]$.[1]

Internal Synchronization: Keep all clocks in the system as close to each other as possible, running at the rate of their physical hardware clocks, except for isolated points where clock values are reset.

Before we describe our results, we first describe what was known prior to this work. We remark that much previous work was done for fault-tolerant clock synchronization, which is beyond the scope of this thesis.

1.2 Previous Work

Different variants of the clock synchronization problem have been the target of a vast amount of research from both practical viewpoint (e.g., [26, 6, 24, 28, 1, 15]) and theoretical viewpoint (e.g., [16, 19, 7, 13, 33, 3], surveys [31, 30] and references therein); the exact definition of the problem depends both on the intended use of the clocks and on the specific underlying system. The large number of variants is justified by the wide spectrum of applications.

One of the popular variants studied theoretically is internal synchronization in the case where all clocks in the system are assumed to run exactly at the rate of real time (we call such clocks *drift-free* hereafter). Lundelius and Lynch [19] consider the case in which there is a communication link between each pair of processors, and message latency bounds are identical for all links in the system. For this case, they present a synchronization algorithm

[1]In this thesis, numbers range over $\mathbf{R} \cup \{\infty, -\infty\}$ unless explicitly indicated otherwise. Square brackets are used to denote intervals, including the case of infinite intervals.

that gives optimal tightness in the worst possible scenario allowable by the system specifications. Halpern *et al.* [13] generalized the results of [19] to networks whose underlying topology is arbitrary, and whose message latency bounds may be different for each link. The main idea in the analysis of [13] is to formulate the problem as a linear program; solving this program, they find the worst case scenario, and an algorithm is presented so that optimal tightness is guaranteed in this case. In [3], Attiya *et al.* observe that the algorithm of [13] always gives the best worst-case tightness, even if the actual execution happens to be more favorable for synchronization than the worst possible. This observation motivates them to generalize the results of [13]; specifically, in [3] they present an algorithm which gives optimal tightness for each specific execution of their system.

The focus in all the above papers [19, 13, 3] is on obtaining bounds in a centralized off-line fashion. Typically, the algorithms can be viewed as consisting of two stages. In the first stage, timing information is gathered at the processors by sending messages over the links. Then a second stage begins, where all the information is sent to one processor; that processor makes the necessary computation, and distributes the results back to the other processors. Only then can each processor adjust its clock.

Practical work is typically more focused on on-line distributed algorithms. Usually, loosely coupled systems use external synchronization algorithms, and tightly coupled systems use internal synchronization. One important protocol for external synchronization is NTP [25, 26], used over the Internet. Another prominent technique in practice is "probabilistic clock synchronization" proposed by Cristian [6]. In this approach, the transmission time of messages is assumed to adhere to some probability distribution, and the transmission times of different messages are assumed to be independent. Under these assumptions, some stochastic guarantees can be made by the synchronization protocol.

1.3 Contents of This Thesis

Our chief objective in this thesis is to acquire better theoretical understanding of clock synchronization. Our first step towards this goal is to define a mathematical model, in which we state our system assumptions precisely, and define the performance criterion by which we measure the quality of the synchronization algorithm. We then abstract executions of systems using a graph theoretic formulation. Using graphs, we state and prove our main characterization of tightness of clock synchronization. From these results, we derive new optimal external synchronization algorithms and a new lower bound on the tightness of internal synchronization. Moreover, we give evidence that indicates that there is no efficient optimal synchronization algorithm that works for arbitrary clock drift bounds and message latency bounds.

In the remainder of this section, we give a more detailed overview of the thesis.

1.3.1 The Setting

Based on the model of *timed input/output automata* of Lynch and Vaandrager [20], we define in Chapter 2 a new formal model, called *mixed automata*. This model enables us to describe systems with local clocks. Using the formalism of mixed automata, we define in Chapter 3 the environment we consider. Intuitively, the main assumptions expressed by our definitions are the following. First, each message, when received (if at all), has a known latency lower bound which is a finite non-negative real number, and a known latency upper bound which is at least the lower latency bound, but it may be infinite. Secondly, each local clock has known finite non-negative lower and upper drift bounds. And thirdly, each execution that satisfies these bounds is possible. We remark that our assumptions include many cases, such as communication links that may lose, re-order, or duplicate messages arbitrarily; systems with broadcast channels; and the case of processor and link crashes.

To facilitate these properties, we assign to the clock synchronization modules a somewhat "passive" part in the system. Our formulation is such that clock synchronization algorithms do not initiate nor delay message transmission and delivery; rather, in our model, message sending is initiated solely by abstract *send modules*, and the clock synchronization algorithm is allowed to pass information only by "piggybacking" on existing message traffic, where we assume that piggybacking is done instantaneously. Thus, the role of a synchronization algorithm can be viewed as limited to the *interpretation* of executions of the environment as they unfold. (Technically, since our definition of executions contains also the real time of occurrence of events, only a *local view* of the execution, which contains just local times of occurrence, is available for computation.) We remark that our model can be viewed as a distributed version of the model considered in [3].

To evaluate the quality of a synchronization algorithm, we define in Chapter 4 a new measure, which may be of independent interest in its own right. Intuitively, our approach is a combination of the execution-specific approach of [3], the competitive analysis approach [32, 23], and the causality partial order of Lamport [16]. Loosely speaking, we call a clock synchronization algorithm *locally K-competitive* if the tightness of its output at any point at any execution is at most K times the best possible tightness among all correct algorithms, given the local view at that point. An algorithm is called *optimal* if it is locally 1-competitive.

1.3.2 A General Theory

The heart of this thesis is a new analysis of clock synchronization problems. Intuitively, we show that even though clock synchronization problems can be formulated as linear programs [13], fortunately they have a much simpler structure, namely distances in a certain graph.

More specifically, in Chapter 5 we introduce a new concept, which we call *synchronization graphs.* Synchronization graphs are weighted, directed graphs derived from system specifications and local views of executions. Since these quantities are locally available for processing, synchronization graphs can be computed by distributed algorithms. The main result of the theory is a characterization of the achievable tightness of synchronization at any execution in terms of distances in the corresponding synchronization graph. An important property of this result is that these distances can be computed on-line in a distributed fashion, thereby giving rise to new algorithmic techniques for optimal synchronization.

Synchronization graphs provide us with a simple and robust concept that deals in a uniform manner with both the uncertainty of transmission times and the uncertainty due to clock drifts. In Chapter 9 we show how to incorporate additional timing information of certain simple types in synchronization graphs. Moreover, we show a simple property of synchronization graphs which is equivalent to the consistency of views with system specifications. This idea can be used to detect faults.

1.3.3 Applications

After proving the general results in Chapter 5, we turn to derive results for specific synchronization tasks. In Chapter 6 we define and analyze the external synchronization problem. In external synchronization, there is a distinguished source processor whose clock is drift-free; each other processor in the system is required to provide, at all times, bounds on the current reading of the source processor. The difference between the bounds is called the *external tightness* of the synchronization at that point. In Chapter 6, we prove a lower bound on the tightness of synchronization at any point, and present a distributed on-line algorithm that meets this bound at all points. This characterization is done for the general setting, where clock drift bounds and message latency bounds are arbitrary. The algorithm for the general case is inefficient. By contrast, we present an efficient algorithm for optimal external synchronization, under the assumption that all clocks in the system are drift-free. We compare our approach with the popular technique of *round-trip* probes, and explain why our approach is superior.

In Chapter 7, we consider the internal clock synchronization problem, where each processor is required to generate a single "tick," and the internal tightness of synchronization in an execution is a bound on the length of real time interval that contains all ticks. Using synchronization graphs, we obtain a lower bound on the achievable internal tightness of synchronization. Our lower bound generalizes known lower bounds for drift-free clocks [19, 13, 3] to the case of drifting clocks. Moreover, our derivation is relatively simple and intuitive.

In Chapter 8, we show a somewhat surprising result regarding the space complexity of optimal synchronization algorithms. We define a certain computational model, in which output values are restricted to be expressed as linear combination of the inputs with integer coefficients (all known algorithms can be expressed this way). In that model, we show that for any external synchronization algorithm there are scenarios that require unbounded space complexity in order to produce optimal output.

The latter result provides strong evidence to the effect that no single algorithm can be efficient, general and optimal at the same time. Practical algorithms must be efficient; the new algorithms we suggest are optimal.

1.4 Significance of the Results

We believe that this thesis contributes to the understanding of clock synchronization in a number of ways.

First, it suggests a new way of looking at the problem, and presents a constructive characterization of achievable tightness. Even though our results indicate that there is no "ultimate solution" for clock synchronization, i.e., an algorithm that is general, efficient and optimal, we believe that using the techniques presented in this thesis, better practical algorithms can be developed, by compromising generality or optimality.

We also believe that the discovery of synchronization graphs is an important contribution to the research of timing-based systems. In some sense, synchronization graphs can be viewed as the extension of Lamport's graphs [16], used to describe executions of completely asynchronous systems, to the case where processors have clocks.

In addition, we think that our approach of local competitiveness can be used for problems in different settings, as it captures an intuitive notion of flexible algorithms that guarantee output close to the best possible for each possible scenario.

1.5 Critique of the Results

Informally, the usefulness of synchronization graphs relies on a few strong assumptions.

(1) The system specification is such that if an event may occur at either of two points, then this event may occur at any point between them.

(2) Processors and communication links follow the system specification.

(3) All executions that satisfy the system specifications are possible.

These assumptions are restrictive. Assumption (1), for example, rules out the case that local clocks run at a fixed but unknown rate. It also rules out systems where message transmission time can be a point in either of two disjoint intervals (this may be the case, for example, when using links that divide the communication into discrete frames). Assumption (2) seems even more problematic: even if the specification allows for some limited kind of faults, it is hardly ever the case that one can guarantee operation of distributed systems without unpredictable faults. Clocks are particularly volatile, as the many papers about fault-tolerant clock synchronization can testify. Assumption (3) seems unrealistic as well: intuitively, it means that all possible timing information is given in the system specification. In many cases, however, additional information can be obtained, e.g., from a human operator.

Let us defend our thesis. The first assumption is absolutely essential for our analysis; the whole theory breaks down if the timing specification is such that there are events that may not occur between points in which they are allowed to occur. We claim, however, that our formulation is appropriate in many cases. For example, when the uncertainty of message transmission times is relatively high, the effect of discrete communication frames is negligible. Also, while conventional quartz clocks (such as the ones used in most CPUs) usually maintain a fixed rate, this rate may change abruptly, thus making the rate look as if it takes values from a continuous range. Hence we argue that assumption (1) seems to be a reasonable abstraction.

Consider assumption (2). For systems with faults, our analysis provides a partial solution in the form of *fault detection*. Even though we do not know how to use synchronization graphs directly to correct errors, we know how to use synchronization graphs to detect them. Moreover, when computing distances over synchronization graphs (as our techniques suggest), the detection comes "for free." It is also conceivable that synchronization graphs can be used in conjunction with some fault tolerance scheme that uses redundancy to eliminate erroneous information.

Assumption (3) is required only for the optimality claims, that is, we use it to obtain lower bounds on the achievable tightness of synchronization. Our algorithms work just as fine if this assumption is removed: it might be the case, however, that additional information can be used to improve performance. Some cases of additional timing information can be modeled by clock synchronization graphs: we give a few simple examples in Chapter 9.

Finally, let us address the validity of our assumption that clock synchronization algorithms are "passive," i.e., that they do not initiate message sending by themselves. We argue that this assumption is not really restrictive; it is used as a convenient theoretical abstraction that enables us to compare different algorithms. Using this model, we view clock synchronization algorithms as if their role is merely to interpret the execution; if an algorithm is optimal in our sense, then it gives the tightest results for *any* execution, and can be used under any pattern of message traffic.

1.6 Structure of this Thesis

The organization of this thesis is as follows. Each chapter begins with a short description of its contents, and ends with an intuitive summary of the main ideas. In Chapter 2 we define the mixed automaton model, which provides us with the formalism we use in describing the systems considered in this thesis. In Chapter 3 we describe the architecture of the clock synchronization systems studied in this thesis, and define the basic notions of views and patterns. In Chapter 4 we define the synchronization tasks we consider, and the way we evaluate their quality, namely the concepts local competitiveness and optimality for synchronization algorithms. In Chapter 5 we define the concept of synchronization graphs, and present our main results. In Chapter 6 we consider the external clock synchronization problem. We give matching bounds on the tightness for general systems, and an efficient optimal algorithm for systems with drift-free clocks. In Chapter 7 we give a lower bound on the achievable tightness for internal synchronization. In Chapter 8 we prove a space lower bound for optimal external synchronization algorithms for general systems. In Chapter 9 we present a few extensions to the concept of synchronization graphs. We conclude in Chapter 10 with a few critical remarks about the results, subsequent work, and open problems.

In Appendix A, we describe the standard method of time-space diagrams. An index is given at the end of the thesis, to aid the reader in tracing definitions of concepts.

Chapter 2

The Mixed Automaton Model

In this chapter we define the mixed automaton model, which is the underlying computational model we consider in this work. Our goal is to formalize the notion of a distributed system with clocks. The development in this chapter is elementary: some readers may wish to skip directly to the more specific definitions of clock synchronization systems in Chapter 3, and refer to the general definitions of this chapter when appropriate.

The mixed automaton model is based on the *timed I/O automata* model of Lynch and Vaandrager [22, 20], abbreviated TIOA henceforth. An important feature of the model is that simple modules, under certain compatibility conditions, can be combined to obtain a more complex module.[1] The main idea in our model, as described in this chapter, is that states of the system contain a component called *now*, which describes the (formal) real time in which the state exists, and components called *local_time*, which describe the readings of the local clocks in that state. (In TIOA, there are no special components for local times.) Time passage is formalized using a special action denoted ν. The *now* and the *local_time* components are changed only by the time-passage action, which means that the local times represent local clocks that cannot be reset.

We open this chapter in Section 2.1 with the definition of mixed automata, and also define a few particular properties of mixed automata that we shall use later. In Section 2.2 we define the notions used to describe how an automaton "runs," namely executions and timed traces. We conclude this chapter by describing composition of mixed automata, which tells us how distinct submodules communicate within a larger module.

[1] We shall use the terms "automaton" and "module" interchangeably throughout this thesis.

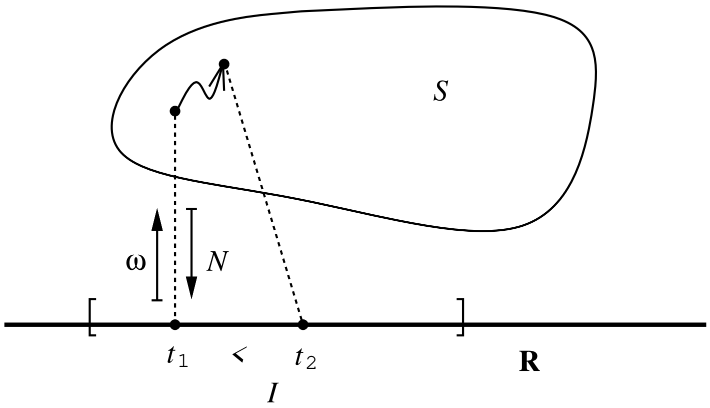

Figure 2-1: *Illustration of Definition 2.1. The N function maps elements of S to real numbers. The trajectory ω is an inverse of N, and maps the "$<$" relation to a "$\rightarrow$" relation.*

2.1 Definition of Mixed Automata

Our first step is to give a definition of *trajectories* (adapted from [20]), which have turned out to be a key concept in the formal analysis of real-time systems (see, e.g., [10, 21]). Intuitively, a trajectory for a given interval will be used to describe an "evolution" of a non-deterministic system when only time passes through that interval of time. The definition below is stated in general terms; the specialization for our purposes is done later. Figure 2-1 gives an illustration of the following definition.

Definition 2.1 *Let S be a set, let N be a function $N : S \mapsto \mathbf{R}$, and let "$\rightarrow$" be a binary relation over S.*[2] *Given a (possibly infinite) interval I of $\mathbf{R}$, a* trajectory *for I, S, N and $\rightarrow$ is a function $\omega : I \mapsto S$, such that $N(\omega(t)) = t$ for all $t \in I$, and such that for all $t_1, t_2 \in I$ with $t_1 < t_2$, we have $\omega(t_1) \rightarrow \omega(t_2)$.*

The interpretation of the abstract notion of trajectory becomes clearer when we define automata. Intuitively, a mixed automaton is a formal representation of a non-deterministic system in a framework of real time, which is represented by non-negative real numbers. In this context, S in Def. 2.1 is used to represent the set of system states; each state s contains the single time point of its existence, which given by a $now(s)$ mapping (corresponding to N in Def. 2.1); a trajectory of an interval is the way the states change while time values range over that interval. Assuming that $\rightarrow$ is a relation (rather than a function) corresponds to the non-deterministic nature of the system.

[2] Throughout this thesis we denote the set of real numbers by $\mathbf{R}$, and the non-negative reals by $\mathbf{R}^+$.

We now proceed with the definition of mixed automata. In addition to the *now* attribute of states which represents real time (as in the TIOA model [20]), a state of a mixed automaton may also have local times attributes, for each local clock. The locations of clocks are represented by special objects called *sites*. Formally, we have the following definition.

Definition 2.2 (Mixed I/O Automata) *A* mixed I/O automaton *A is defined by the following components.*

- *A finite, possibly empty set of* sites *sites(A).*
- *A set of* states *states(A) with the following mappings:*

$$\begin{aligned} now &: states(A) \mapsto \mathbf{R}^+ \\ \overline{\boldsymbol{T}} &: sites(A) \times states(A) \mapsto \mathbf{R}^{|sites(A)|} \end{aligned}$$

  The value $now(s)$ is called the real time *of s. For a site $v \in sites(A)$, we use the notation $local\_time_v(s) = \overline{\boldsymbol{T}}(v, s)$. $\overline{\boldsymbol{T}}(s)$ is used as a function from sites to $\mathbf{R}$.*

- *A nonempty set of* start states *$start(A) \subseteq states(A)$.*
- *A set $acts(A)$ of* actions. *One of the actions is a special* time-passage *action, denoted ν; the other actions are called* discrete. *The actions are partitioned into* external *and* internal *actions, where time passage is considered to be external. The* visible *actions are the discrete external actions. Visible actions are partitioned into* input *and* output *actions.*
- *A* transition *relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$. We also use the shorthand $s \xrightarrow{\pi}_A s'$ for $(s, \pi, s') \in trans(A)$; when the context is clear, we sometimes write $s \xrightarrow{\pi} s'$. For an action π and a state s, if there exists a state s' such that $s \xrightarrow{\pi} s'$, then we say that π is* enabled *in s.*

We require that A satisfy the following axioms.

C1 *For all $s \in start(A)$, $now(s) = 0$.*

C2 *For all $s \xrightarrow{\pi} s'$ with $\pi \neq \nu$, $now(s) = now(s')$ and $\overline{\boldsymbol{T}}(s) = \overline{\boldsymbol{T}}(s')$.*

C3 *For all $s \xrightarrow{\nu} s'$, $now(s') > now(s)$.*

C4 *If $s \xrightarrow{\nu} s'$ and $s' \xrightarrow{\nu} s''$, then $s \xrightarrow{\nu} s''$.*

C5 *For all $s \xrightarrow{\nu} s'$, there exists a trajectory ω for $[now(s), now(s')]$, the state set, the now mapping and the time passage subrelation $\{(s, \nu, s') \in trans(A)\}$, such that $\omega(now(s)) = s$ and $\omega(now(s')) = s'$.*

When we talk about more than a single automaton, we use subscripts to denote the context. For example, $local\_time_{A,v}$ denotes the local time function of automaton A at site v.

We remark that timed I/O automata, as defined in [20], are a special case of mixed automata, where the site set is empty.[3]

Example: the SENDER **automaton.** Let us illustrate the concept of a mixed automaton with a toy example, which we shall return to later. We define an automaton, called SENDER, that has a single input action called *Receive_Message*, and a single output action called *Send_Message*. The SENDER automaton is equipped with a local clock that runs at the rate of real time; the behavior of SENDER is very simple: it may output *Send_Message* only if there was at least one *Receive_Message* input since the previous *Send_Message* output. The following is a formal description of SENDER.

- There is a single site, which we choose to call v (any other name can do as well).
- The state set is $\Big\{(t, T, pend) \,:\, t \in \mathbf{R}^+, T \in \mathbf{R}, pend \in \{\text{TRUE}, \text{FALSE}\}\Big\}$. For a state $s = (t, T, pend)$, we have $now(s) = t$, $\overline{\boldsymbol{T}}(s) = (T)$, and $local\_time_v(s) = T$. In words, the real time of $(t, T, pend)$ is t, and the local time of $(t, T, pend)$ at site v is T. The Boolean flag *pend* will be used to indicate whether there is a "pending output" (see below).
- The set of start states is $\{(0, T, \text{TRUE}) : T \in \mathbf{R}\}$, i.e., all states with real time 0 and $pend = \text{TRUE}$. This definition means that the initial local time at v is arbitrary, and that *Send_Message* may be the first action of SENDER.
- The set of actions is $\{\nu, Receive\_Message, Send\_Message\}$, where ν is the time passage action, *Receive_Message* is a discrete input action, and *Send_Message* is a discrete output action. Hence both *Receive_Message* and *Send_Message* are external and visible.

[3]The converse is also true: given a mixed automaton, one can model it as a particular kind of TIOA.

Sites: *a single site* v

State:

now: *a non-negative real number, initially* 0
local_time: *a real number, initially arbitrary*
pend: *a Boolean flag, initially* TRUE

Actions:

Receive_Message *(input)*
Pre: *none*
Eff: $pend \leftarrow$ TRUE

Send_Message *(output)*
Pre: $pend =$ TRUE
Eff: $pend \leftarrow$ FALSE

ν : *(time passage)*
Pre: $b > 0$
Eff: $now \leftarrow now + b$
$local\_time \leftarrow local\_time + b$

Figure 2-2: SENDER: *an example of a mixed automaton.*

- The transition relation is as follows.

  First, for all $t \geq 0$, $T \in \mathbf{R}$, $pend \in \{\text{TRUE}, \text{FALSE}\}$ and $b > 0$, we have $(t, T, pend) \xrightarrow{\nu} (t + b, T + b, pend)$. This means that time passage is always enabled, and that the local time is increased exactly by the amount of real time that passes.

  Secondly, for $pend \in \{\text{TRUE}, \text{FALSE}\}$, $((t, T, pend), Receive\_Message, (t, T, \text{TRUE}))$ is a transition. This means that the *Receive_Message* action is always enabled, and its effect is to set *pend* to TRUE.

  Finally, we have that $((t, T, \text{TRUE}), Send\_Message, (t, T, \text{FALSE}))$ is a transition, which means that the *Send_Message* action is enabled exactly at all states where $pend =$ TRUE, and its effect is to set $pend =$ FALSE.

Formal description of automata will usually be done in this thesis using the "precondition-effect" notation given in Figure 2-2. This more structured representation will be sufficient to describe the algorithms we study. When the "Pre" clause is omitted from the description of a transition, the interpretation is that the action is always enabled. ∎

2.1.1 Projections, Equivalent Automata

In this section we define the technical notions of projection and equivalent automata.

Intuitively, a projection of an automaton on one of its sites is the restriction of the automaton to describe only the clock of that site.

Definition 2.3 *The* projection *of a mixed automaton A on a site $v \in sites(A)$, or the* clock of A at v*, denoted by $A|_v$, is the mixed automaton defined as follows.*

- $sites(A|_v) = \{v\}$.
- $acts(A|_v) = \{\nu\}$.
- *For a state $s \in states(A)$, let $s|_v$ be the pair $(now_A(s), \overline{\boldsymbol{T}}_A(v,s))$. With this notation, we have*
  - $states(A|_v) = \{s|_v : s \in states(A)\}$*, and we set*

$$
\begin{aligned}
now_{A|_v}(s|_v) &= now_A(s) \\
\overline{\boldsymbol{T}}_{A|_v}(v, s|_v) &= \overline{\boldsymbol{T}}_A(v,s)
\end{aligned}
$$

  - $start(A|_v) = \{s|_v : s \in start(A)\}$.
  - $trans(A|_v) = \{(s|_v, \nu, s'|_v) : (s, \nu, s') \in trans(A)\}$.

We have the following lemma.

Lemma 2.1 *For any mixed automaton A, for all $v \in sites(A)$,$A|_v$ is a mixed automaton.*

Proof: By inspection of the axioms. ∎

We conclude this section with a definition of equivalent automata. Intuitively, two automata are equivalent if they are the same, up to renaming and multiplicity of equivalent states. Formally, we have the following definition.

Definition 2.4 *A mixed automaton B is said to* extend *a mixed automaton A if $sites(A) \subseteq sites(B)$, $acts(A) \subseteq acts(B)$, and there exists a mapping $f : states(B) \mapsto states(A)$ such that the following conditions hold for all $s \in states(B)$.*

- $now_A(f(s)) = now_B(s)$.

- *For all $v \in sites(A)$, $local\_time_{A,v}(f(s)) = local\_time_{B,v}(s)$.*
- *$f(s) \in start(A)$ iff $s \in start(B)$.*
- *For all $\pi \in acts(A)$, we have $(f(s), \pi, f(s')) \in trans(A)$ iff $(s, \pi, s') \in trans(B)$.*

A and B are said to be equivalent*, denoted $A \equiv B$, if A extends B and B extends A.*

2.1.2 Clock Types

In this work, we shall study automata where local clocks have bounded drifts, as defined below.

Definition 2.5 *Let v be a site of a given mixed automaton A. If there exist $0 < \underline{\varrho} \leq \overline{\varrho} < \infty$ such that for all all $s \xrightarrow{v} s'$,*

$$\underline{\varrho}(now(s') - now(s)) \;\leq\; local\_time_v(s') - local\_time_v(s) \;\leq\; \overline{\varrho}(now(s') - now(s)) \;,$$

*then $A|_v$ is called a $(\underline{\varrho}, \overline{\varrho})$-*clock*. A clock $A|_v$ is called a* bounded-drift clock *if it is a $(\underline{\varrho}, \overline{\varrho})$-clock for some $0 < \underline{\varrho} \leq \overline{\varrho} < \infty$. A $(1,1)$-clock is also said to be* drift-free*.*

Alternatively, one can think of a clock as a collection of real-valued "clock functions" $\{T(t)\}$, where t denotes real time. In this representation, a $(\underline{\varrho}, \overline{\varrho})$-clock consists of functions $T(t)$ such that $\underline{\varrho}(t-t') \leq T(t) - T(t') \leq \overline{\varrho}(t-t')$ for all $t \geq t' \geq 0$ (which also means that all clock functions of a bounded drift clock are continuous), and a drift-free clock is a function of the type $T(t) = t + a$ for some constant a. We formalize this interpretation in Definition 2.12, after we define executions.

2.1.3 Real Time Blindness

In our model, real time is a part of the state of the system. In many systems, access to real time is restricted to occur only via special physical devices, such as clocks. To model this property, we introduce the notion of real-time blindness in the following definition. The definition is specialized for bounded-drift clocks.

Definition 2.6 *Let A be a mixed automaton such that each $v \in sites(A)$ is a $(\underline{\varrho}_v, \overline{\varrho}_v)$-clock. A is said to be* real-time blind for $(\underline{\varrho}_v, \overline{\varrho}_v)$ *if there exists an equivalent automaton $A' \equiv A$, with* *a set $B(A')$ and a mapping basic : $states(A') \mapsto B(A')$ such that the following conditions are satisfied.*

- *For all $b \in B(A')$, all mappings $\overline{T} : sites(A') \mapsto \mathbf{R}$ and all $t \in \mathbf{R}^+$, there exists $s \in states(A')$ such that $basic(s) = b$, $now(s) = t$ and $\overline{\boldsymbol{T}}(v, s) = \overline{T}(v)$ for all $v \in sites(A')$.*
- *For all $s_1 \xrightarrow{\nu} s_2$, $basic(s_1) = basic(s_2)$.*
- *For all $s_1, s_2, s_1', s_2' \in states(A')$: if $(s_1, \pi, s_2) \in trans(A')$ for $\pi \neq \nu$, and*

$$\begin{aligned} \overline{\boldsymbol{T}}(s_1) &= \overline{\boldsymbol{T}}(s_1') \\ basic(s_1') &= basic(s_1) \\ basic(s_2') &= basic(s_2) \end{aligned}$$

  then $(s_1', \pi, s_2') \in trans(A')$.

- *For all $s_1, s_2, s_1', s_2' \in states(A')$: suppose $(s_1, \nu, s_2) \in trans(A')$, and let $\Delta = now(s') - now(s)$. If for all $v \in sites(A')$ we have*

$$\begin{aligned} basic(s_1') &= basic(s_1) \\ \overline{\boldsymbol{T}}(s_1') &= \overline{\boldsymbol{T}}(s_1) \\ \overline{\boldsymbol{T}}(s_2') &= \overline{\boldsymbol{T}}(s_2) \\ \overline{\boldsymbol{T}}(v, s_2') - \overline{\boldsymbol{T}}(v, s_1') &\in \left[\underline{\varrho}_v \cdot \Delta \ , \ \overline{\varrho}_v \cdot \Delta\right] \end{aligned}$$

  then $(s_1', \nu, s_2') \in trans(A')$.

Intuitively, an automaton is real-time blind if each of its states can be decomposed into three components, called the real time, the local times, and the basic component. We require that this decomposition is such that time passage action has no effect on the basic component, and that the enabledness of actions is independent of the real time component. The time passage action is special, since the clock drift bounds imply that the local times component and the real time component are related. In this case we therefore require that all amounts of real time passage allowed by the drift bounds are possible by a real-time blind automaton.

Example. It is easy to verify that SENDER is real time blind for $(1, 1)$: the decomposition of its states is readily given. Specifically, a state $(t, T, pend)$ has real time component t, local time component T, and basic component *pend*. Let us verify the properties of this decomposition:

- The state set is $\mathbf{R}^+ \times \mathbf{R} \times \{\text{TRUE}, \text{FALSE}\}$.
- The value of *pend* is never changed by time passage.
- Changes in the value of *pend* depend only its value and the type of action taken.
- Time passage does not depend on the value of the *now* component neither in being enabled nor in the amount of time that passes, except for that the real time may be increased exactly by the amount local time is increased by.

∎

2.1.4 Quiescent States

The following definition formalizes the notion of "idle state," in which nothing happens, and nothing will happen, unless some input occurs.

Definition 2.7 *A state $s \in states(A)$ for some mixed automaton A is called* quiet *if the only actions enabled in s are input actions and time-passage actions. A quiet state s_0 is said to be* quiescent *if the following conditions hold.*

(1) For all $t > 0$ there exists a transition $s_0 \xrightarrow{\nu} s$ such that $now(s') = now(s) + t$.

(2) For all states s such that $s_0 \xrightarrow{\nu} s$, s is quiet.

Intuitively, a state is quiet if the automaton is not poised at doing something at present, and a state is quiescent if the automaton is not intending to do something at the future. An important consequence of quiescence will be proved in Lemma 3.1, in the next chapter.

Example. Examining SENDER once again, we see that all the states of the form (t, T, FALSE) are quiescent: only input and time-passage actions are enabled in them, and only other states of the same form are reachable from them by time passage. ∎

2.2 Executions and Timed Traces

In this section we formalize the concept of system execution and its derivative notions. We remark that the definition of executions of mixed automata we give here is a straightforward extension of the definition of timed executions in [20]. We shall use the following notations (cf. Definition 2.1 and Figure 2-1).

Notation 2.8 *Let I be a (possibly infinite) interval of $\mathbf{R}^+$, and let A be a mixed automaton. A* trajectory *on I of A is a trajectory for I, states(A), the now mapping, and the time-passage relation $\{(s, \nu, s') \in trans(A)\}$. Let ω be a trajectory on I of A. Denote f_now$(\omega) = \inf(I)$, and l_now$(\omega) = \sup(I)$. If I is left-closed, let f_state(ω) denote $\omega($f_now$(\omega))$, and if I is right-closed, let l_state(ω) denote $\omega($l_now$(\omega))$.*

We start with the definition of execution fragments.

Definition 2.9 *Let A be a mixed automaton. An* execution fragment *of A is an alternating (finite or infinite) sequence $\langle \omega_0 \pi_1 \omega_1 \pi_2 \omega_2 \ldots \rangle$ such that*

(1) Each ω_j is a trajectory, and each π_j is a discrete action.

(2) If the sequence is finite, then it ends with a trajectory.

(3) If ω_j is not the last trajectory in the sequence, then its domain is a closed interval. If there is a last trajectory, then its domain is left-closed.

(4) If ω_j is not the last trajectory, then l_state$(\omega_j) \xrightarrow{\pi_{j+1}}$ f_state(ω_{j+1}).

The *duration* of a finite execution fragment $\langle \omega_0 \pi_1 \omega_1 \pi_2 \omega_2 \ldots \omega_N \rangle$ is the (possibly infinite) interval $[$*f_now*$(\omega_0),$ *l_now*$(\omega_N)]$. The duration of an infinite execution fragment $\langle \omega_0 \pi_1 \omega_1 \pi_2 \omega_2 \ldots \rangle$ is the interval $[$*f_now*$(\omega_0), \sup_i$ *l_now*$(\omega_i)]$.

Definition 2.10 *An* execution *of a mixed automaton A is an execution fragment $\langle \omega_0 \pi_1 \omega_1 \pi_2 \omega_2 \ldots \rangle$ of A such that f_state$(\omega_0) \in start(A)$.*

Call an execution *admissible* if its duration is infinite. In this work we consider only *feasible* automata, defined by the condition that each finite execution of a feasible automaton can be extended to an admissible execution.

Given an execution fragment $\langle \omega_0 \pi_1 \omega_1 \ldots \rangle$, we define for each event π_i its *times of occurrence*, $\overline{\boldsymbol{T}}(\pi_i) = \overline{\boldsymbol{T}}($*l_state*$(\omega_{i-1}))$ (thus $\overline{\boldsymbol{T}}(\pi_i)$ is a mapping that assigns to each site a local time). Sometimes actions will be associated with a single site. If a step π is associated with a site v, we refer to *the local time of occurrence* of π, defined by $local\_time(\pi) = \overline{\boldsymbol{T}}(\pi)(v)$. The *real time of occurrence* is defined to be $now(\pi_i) = now(l\_state(\omega_{i-1}))$.

Next, we define the notion of timed traces.

Definition 2.11 *Given a finite execution fragment $e = \langle \omega_0 \pi_1 \omega_1 \ldots \omega_N \rangle$, the* timed trace *of e is a triple $((t_s, \overline{\boldsymbol{T}}_s), \alpha, (t_f, \overline{\boldsymbol{T}}_f))$, where the* start time *is $\overline{\boldsymbol{T}}_s = \overline{\boldsymbol{T}}(f\_state(\omega_0))$ and $t_s = now(f\_state(\omega_0))$; the* finish time *is $\overline{\boldsymbol{T}}_f = \overline{\boldsymbol{T}}(l\_state(\omega_N))$ and $t_f = now(l\_state(\omega_N))$;[4] and α is a sequence of triples $(\pi_i, t_i, \overline{T}_i)$, where $\pi_1, \pi_2 \ldots$ is the sequence of all visible events in the execution, and for each i, t_i is the real time of occurrence of π_i, and $\overline{T}_i$ is the times of occurrence of π_i. For an infinite execution fragment, finish time is given by $t_f = \sup_{\omega_i, t}(now(\omega_i(t)))$, and $\overline{\boldsymbol{T}}_f(v) = \sup_{\omega_i, t}(local\_time_v(\omega_i(t)))$ for each site v.*

We close this section with a definition of the natural concept of clock function.

Definition 2.12 (Clock Functions) *Let $e = \langle \omega_0 \pi_1 \ldots \rangle$ be an execution of an automaton A, and let $v \in sites(A)$. The* clock function *of v in e is a mapping $local\_time_v : \mathbf{R}^+ \mapsto \mathbf{R}$ such that for all $t \geq 0$, if $t \in [f\_now(\omega_i), l\_now(\omega_i)]$, then $local\_time_v(t) = \overline{\boldsymbol{T}}(\omega_i(t), v)$.*

Recall that the notation *local_time* is also defined as a function from states to the reals; the interpretation being used should be clear from the context.

Finally, given an automaton A and a site $v \in sites(A)$, we define the *set of clock functions* of v to consists of all clock functions of the projected automaton $A|_v$.

2.3 Composition of Mixed Automata

We now proceed to define the composition of mixed automata. First, we define composition of states.

Definition 2.13 *Let A and B be mixed automata. Two states $s_A \in states(A)$ and $s_B \in states(B)$ are* compatible *if $now(s_A) = now(s_B)$ and $local\_time_v(s_A) = local\_time_v(s_B)$ for all $v \in sites(A) \cap sites(B)$. The* composition *of two compatible states s_A and s_B, is the pair (s_A, s_B), which has the following attributes.*

[4] Again, note that $\overline{\boldsymbol{T}}_s$ and $\overline{\boldsymbol{T}}_f$ are mappings that assign a local time to each site.

- $now(s_A, s_B) = now(s_A)$.
- *For each site* $v \in sites(A) \cup sites(B)$,

$$\overline{\boldsymbol{T}}(v,(s_A,s_B)) = local\_time_v(s_A,s_B) = \begin{cases} local\_time_v(s_A), & \text{if } v \in sites(A)\ , \\ local\_time_v(s_B), & \text{if } v \in sites(B)\ . \end{cases}$$

For a composed state (s_A, s_B)*, we denote* $(s_A, s_B)|_A = s_A$*, and* $(s_A, s_B)|_B = s_B$*.*

Note that by the compatibility condition, $local\_time_v(s_A \times s_B)$ is well defined for $v \in sites(A) \cap sites(B)$.

We now define a necessary condition for composing mixed automata. We use the notion of projection here (cf. Definition 2.3).

Definition 2.14 *Let* A, B *be two mixed I/O automata. A and B are said to be* compatible *if their output actions are disjoint, the set of internal actions of A is disjoint from the set of all actions of B, and the set of internal actions of B is disjoint from the set of all actions of A. In addition, we require that for all* $v \in sites(A) \cap sites(B)$*, we have that* $A|_v \equiv B|_v$*.*

We are now ready to define composition of automata.

Definition 2.15 (Mixed Automata Composition) *Let A and B be two compatible mixed I/O automata. The* composition $A \times B$ *of A and B is a mixed I/O automaton defined as follows.*

- *The* sites *of* $A \times B$ *are* $sites(A \times B) = sites(A) \cup sites(B)$*.*
- *The* states *of* $A \times B$ *is the set of all compatible pairs of states from* $states(A)$ *and* $states(B)$*.*
- *The* start *set of* $A \times B$ *is the set obtained by composing all compatible pairs of states from* $start(A)$ *and* $start(B)$*.*
- *The set of* actions *of* $A \times B$ *is the union of* $acts(A)$ *and* $acts(B)$*. A discrete action is* external *in* $A \times B$ *exactly if it is external at either A or B, and likewise for* internal actions *of* $A \times B$*. A visible action of* $A \times B$ *is an* output action *if it is an output action of exactly one of either A or B, and it is* input *otherwise.*

- *For any action* $\pi \in acts(A \times B)$ *and states* $s, s' \in states(A \times B)$, *we have* $(s, \pi, s') \in trans(A \times B)$ *iff both the following hold.*

  (1) If $\pi \in acts(A)$ *then* $(s|_A, \pi, s'|_A) \in trans(A)$, *otherwise* $s|_A = s'|_A$.

  (2) If $\pi \in acts(B)$ *then* $(s|_B, \pi, s'|_B) \in trans(B)$, *otherwise* $s|_B = s'|_B$.

Composition defines the way two automata interact: this is done by shared actions. The compatibility condition prohibits shared output actions, or interfering with internal actions of each other, and requires that shared portions of the state have the same underlying structure.

Below we state the basic property of composition.

Lemma 2.2 *If* A *and* B *are compatible mixed I/O automata, then* $A \times B$ *is a mixed I/O automaton.*

Proof: Straightforward. ∎

Notice that we can compose any finite number of compatible automata, by applying the binary composition operator defined above iteratively. The set of executions of the resulting automaton is essentially the same (up to a natural isomorphism), regardless of the order of composition.

We now turn to look at executions of composed automata. The following two lemmas establish connections between executions of a composed automaton and the execution of its constituent automata. First, for an execution e of a composed automaton $A \times B$, let $e|_A$ denote the sequence obtained from e by mapping each state s of e into $s|_A$, omitting all actions of B from e, and for each action π_i of B in e, we merge the resulting trajectories ω_i and ω_{i+1}. Analogously we define $e|_B$. The sequences $e|_A$ and $e|_B$ are called the *projection of* e to A and B, respectively. We have the following simple property for projection of execution of a composed automaton.

Lemma 2.3 *Let* e *be an execution of a composed automaton* $A \times B$. *Then* $e|_A$ *and* $e|_B$ *are executions of* A *and* B, *respectively.*

Proof: Immediate from the definitions. ∎

We now prove a converse for Lemma 2.3. To be able to state it, we have to make a few technical definitions. Fix a mixed automaton A. A *times form* for a set of sites

$V \subseteq sites(A)$ is a mapping $\overline{\boldsymbol{F}} : V \mapsto \mathbf{R}$. A *timed sequence* for A is a sequence $\sigma = \langle(\pi_1, now(\pi_1), \overline{\boldsymbol{F}}_{\pi_1}), (\pi_2, now(\pi_2), \overline{\boldsymbol{F}}_{\pi_2}\rangle$, where each π_i is a visible action of A, $now(\pi_i)$ is a non-negative number, and $\overline{\boldsymbol{F}}_{\pi_i}$ is a times form. We require that the sequence $\langle now(\pi_i)\rangle_{i\geq 1}$ is non-decreasing. A *form* for A is a triple $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))$, where σ is a timed sequence of A; t_s and t_f are non-negative real numbers called the start and finish real time, respectively; and $\overline{\boldsymbol{F}}_s$ and $\overline{\boldsymbol{F}}_f$ are times forms, called the start and finish times forms, respectively. Notice that for a given automaton, every timed trace is a form; the converse, however, is not true in general, since a form for A need not be obtained from an execution of A.

Let $\overline{\boldsymbol{F}}$ be a times form for a site set V. The *projection* $\overline{\boldsymbol{F}}|_{V'}$ of $\overline{\boldsymbol{F}}$ for $V' \subseteq V$ is obtained by restricting the domain of $\overline{\boldsymbol{F}}$ to sites in V' only. Given a timed sequence for a composed automaton $A \times B$, its projection $\sigma|_A$ is defined as the subsequence of actions of A, where the times form for each action is projected on $sites(A)$. Finally, the *projection of a form* for a composed automaton is obtained by projecting the start times form, the timed sequence, and the finish times form, i.e, $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f\overline{\boldsymbol{F}}_f))|_A = \Big((t_s, \overline{\boldsymbol{F}}_s|_{sites(A)}), \sigma|_A, (t_f, \overline{\boldsymbol{F}}_f|_{sites(A)})\Big)$.

In the following lemma we prove that a converse to Lemma 2.3 is also true, i.e., if we have executions of A and of B that are compatible in a certain sense, then there exists an execution of $A \times B$ that, after projections, looks like either of the given executions (of A and of B).

Lemma 2.4 *Let $A \times B$ be the composition of compatible mixed automata A and B, and let $((t_s, \overline{\boldsymbol{T}}_s), \sigma, (t_f, \overline{\boldsymbol{T}}_f))$ be a form for $A \times B$. Suppose that there exist execution fragments of A and B whose timed traces are the projection of $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))$ on A and on B, respectively, and such that for all $v \in sites(A) \cap sites(B)$ we have $local\_time_{A,v}(t) = local\_time_{B,v}(t)$ for all $t \in [t_s, t_f]$. Then there exists an execution fragment of $A \times B$ whose timed trace is $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))$.*

Proof: Suppose $\sigma = \langle \pi_1, \pi_2, \ldots\rangle$, $\sigma|_A = \langle \pi_{i_1}, \pi_{i_2}, \ldots\rangle$, and $\sigma|_B = \langle \pi_{j_1}, \pi_{j_2}, \ldots\rangle$. By the assumption, we can "fill in" trajectories ω_{i_l} and ω_{j_m} such that the following properties hold (see Figure 2-3 for an example).

(1) The alternating sequence $e_A = \langle \omega_{i_0}\pi_{i_1}\omega_{i_1}\pi_{i_2}\ldots\rangle$ is an execution fragment of A, and the alternating sequence $e_B = \langle \omega_{j_0}\pi_{j_1}\omega_{j_1}\pi_{j_2}\ldots\rangle$ is an execution fragment of B.

(2) The timed trace of e_A is $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))|_A$, and the timed trace of e_B is $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))|_B$.

(3) For all sites $v \in sites(A) \cap sites(B)$ and $t \in [t_s, t_f]$, $local\_time_{A,v}(t) = local\_time_{B,v}(t)$.

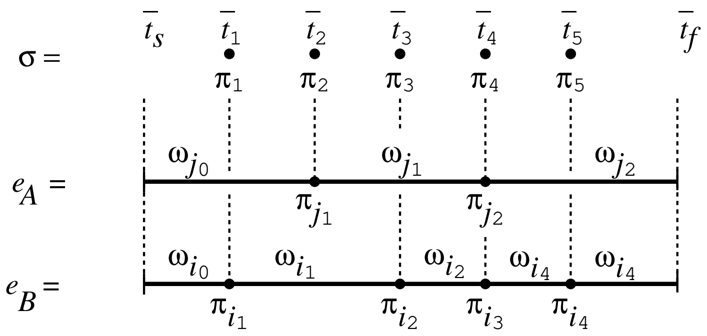

Figure 2-3: *An example for the scenario considered in the proof of Theorem 2.4. While σ is a form for $A \times B$, e_A and e_B are executions of A and B whose timed traces are $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))|_A$ and $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))|_B$, respectively.*

Using these trajectories, we construct an execution of $A \times B$ in a piecewise fashion. For ease of notation, let us define $r_k = now(\pi_k)$, and $r_0 = t_s$. We now show how to construct a trajectory ω_k for the time interval $[r_k, r_{k+1}]$, where $k \geq 0$. Let i_l, j_m be the greatest indices such that π_{i_l} and π_{j_m} occur before π_{k+1} in σ, or 0 if no such events exist. Define r_{i_l} to be the *now* value of π_{i_l}, or t_s if $i_l = 0$; define r_{j_m} analogously. (Notice that r_k is the maximum of r_{i_l} and r_{j_m}.) For example, in Figure 2-3 and with $k = 3$, we have $i_l = i_2$ and $j_m = j_1$.

We define ω_k using ω_{i_l} and ω_{j_m} using state composition, namely $\omega_k(t) = \omega_{i_l}(t) \times \omega_{j_m}(t)$. We claim that ω_k is a trajectory on $[r_k, r_{k+1}]$ for $A \times B$. We prove this as follows. First, for all $t \in [r_k, r_{k+1}]$, $now_A(\omega_{i_l}(t)) = now_B(\omega_{j_m}(t)) = t$, and for all $v \in sites(A) \cap sites(B)$ we have by assumption that $local\_time_{A,v}(t) = local\_time_{B,v}(t)$. It follows that $\omega_{i_l}(t) \times \omega_{j_m}(t)) \in states(A \times B)$ for all t in the interval. Secondly, let $r_k \leq t_1 < t_2 \leq r_{k+1}$. By the properties of A and B, respectively, we have that $(\omega_{i_l}(t_1), \nu, \omega_{i_l}(t_2)) \in trans(A)$, and $(\omega_{j_m}(t_1), \nu, \omega_{j_m}(t_2)) \in trans(B)$. Also, for all $v \in sites(A) \cap sites(B)$ we have by assumption that $local\_time_{A,v}(t_1) = local\_time_{B,v}(t_1)$ and $local\_time_{A,v}(t_2) = local\_time_{B,v}(t_2)$. It therefore follows that $(\omega_{i_l}(t_1) \times \omega_{j_m}(t_1)\,, \nu,\, \omega_{i_l}(t_2) \times \omega_{j_m}(t_2)) \in trans(A \times B)$, showing that ω_k is a trajectory for $A \times B$.

To complete the construction, we need to combine the trajectories by the visible actions of σ. But this immediately follows since for $k > 0$, $(l\_state(\omega_{k-1}), \pi_k, f\_state(\omega_k)) \in trans(A \times B)$ by definitions. We conclude by noting that the execution fragment constructed above agrees with the time forms $(t_s, \overline{\boldsymbol{F}}_s)$ and $(t_f, \overline{\boldsymbol{F}}_f)$. ∎

Corollary 2.4.1 *Let $A_1 \times A_2 \times \cdots A_n$ be the composition of compatible mixed automata $A_1, \ldots, A_n$, and let $((t_s, \overline{\boldsymbol{T}}_s), \sigma, (t_f, \overline{\boldsymbol{T}}_f))$ be a form for $A_1 \times A_2 \times \cdots A_n$. Suppose that for $i = 1, \ldots, n$ there exist execution fragments of A_i whose timed traces are the projection of $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))$ on A_i. Suppose further that if $v \in sites(A_i) \cap sites(A_j)$ for some i, j, then we have $local\_time_{A_i,v}(t) = local\_time_{A_j,v}(t)$ for all $t \in [t_s, t_f]$. Then there exists an execution fragment of $A_1 \times A_2 \times \cdots A_n$ whose timed trace is $((t_s, \overline{\boldsymbol{F}}_s), \sigma, (t_f, \overline{\boldsymbol{F}}_f))$.*

Proof: By applying Theorem 2.4 to A_1 and A_2, and then to $A_1 \times A_2$ and A_3 etc. ∎

Summary

In this chapter we defined the *mixed automaton* model, which is the underlying computational model we shall consider in the remainder of this work. The mixed automaton model is based on the timed I/O automata model of Lynch and Vaandrager [22, 20]. Our model formalizes the notion of a system with local clocks. We defined the basic notions of *executions* and their *timed traces*, which roughly are the sequences of input and output events in executions. We made a few notational conventions, described intuitively as follows.

- Clock locations are called *sites*.
- The *real time of occurrence* of an event π is denoted by $now(\pi)$.
- For a site v and an event π, $local\_time_v(\pi)$, is the *local time of occurrence* of π, defined by the value of the clock of v when π occurs.
- A *bounded-drift clock* is a clock whose rate of progress with respect to real time is bounded by a *drift lower bound* and a *drift upper bound.* A $(\underline{\varrho}, \overline{\varrho})$*-clock* is a bounded drift clock with drift bounds $0 \le \underline{\varrho} \le \overline{\varrho}$. A $(1,1)$-clock is called a *drift-free clock.*
- An automaton is *real-time blind* if it cannot access the real time component of the state. (It may access the local time component.)
- A state is *quiescent* if no locally-controlled action is enabled in it, and no such action will become enabled by time passage alone.

An important feature of the model is that simple modules, under certain compatibility conditions, can be combined to obtain a more complex module.

Chapter 3

Clock Synchronization Systems

In this chapter we use the formalism developed in Chapter 2 to describe the clock synchronization systems we shall be studying. The main idea in the system definition in this chapter (first introduced by Attiya *et al.* [3]) is to partition the system into two: an active part (called environment) that generates messages and delivers them, and a passive part, played by the clock synchronization algorithm, whose role is to interpret the resulting communication patterns. This is in contrast to conventional viewpoints, where synchronization algorithms may initiate the sending of a message. Intuitively, in our framework algorithms have to work with any possible message traffic generated by the environment.

This chapter in organized as follows. In Section 3.1 we carefully define the system, by describing each of its basic components and the way they interact. This modeling is intended to be reasonably close to the way systems are constructed, e.g., it includes definitions of processors and communication links.

In Section 3.2 we shift our standpoint to a more conceptual one: we isolate the role of the synchronization algorithm versus an adversarial *environment*, which controls the local clocks, and message send and receive events. We define the key notions of the *view* and the *pattern* of an execution of a clock synchronization system, which describe the information in the execution which is relevant for clock synchronization tasks. These notions are defined with respect to an execution of the system. To capture the properties of distributed on-line system (discussed in Chapter 4), we also define the notion of *local view* of an execution, which is the part of the view which can be known at a processor at a time point.

We conclude the system model chapter in Section 3.3, where we prove the basic property

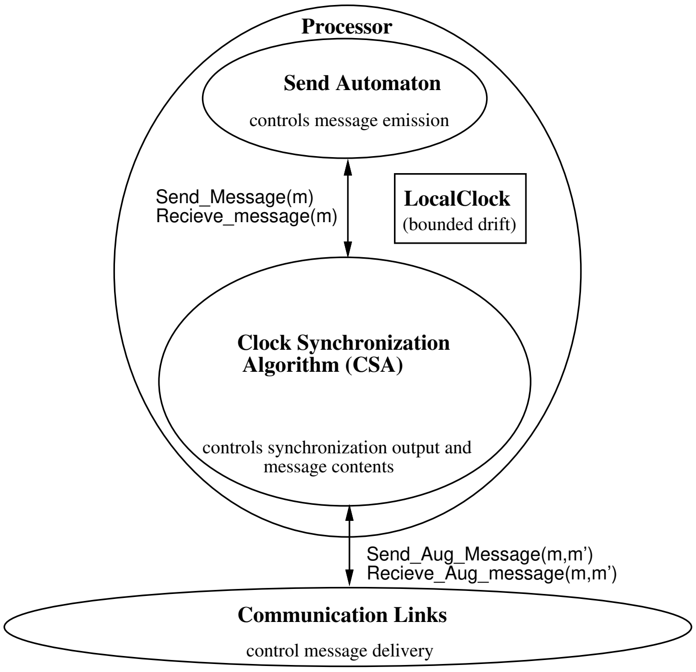

Figure 3-1: *The automata and interfaces at one node of a clock synchronization system. Each processor has a local clock; only the send modules initiate message sending. The clock synchronization modules must work using piggybacking on existing traffic.*

used in lower-bound arguments in this thesis. Intuitively, this property is that (1) all executions that satisfy the timing specification of the system are possible, and (2) the output of a synchronization algorithm depends only on the view of the execution, which contains local times of events, but no real times.

3.1 Specifications of System Components

The system has an *underlying graph*, which is a directed graph whose nodes represent processors and whose edges represent unidirectional communication links. We call the nodes of the underlying graph *processors*, to avoid confusion with nodes of other kinds of graphs defined later.

Roughly speaking, the system we describe is as follows (see Figure 3-1). Each processor has a bounded-drift clock (cf. Definition 2.5). Processors communicate by sending messages over the links. Message sends are initiated only by the send modules, in an arbitrary fashion (i.e., a send action can be taken at any time). The clock synchronization algorithm (abbreviated CSA henceforth) can only piggyback messages on the existing traffic in order to carry out the specific synchronization task at hand.

In our notation, send modules output $\mathit{Send\_Message}(m)$ actions. For each $\mathit{Send\_Message}(m)$ action at a processor v, the CSA at v must immediately output a $\mathit{Send\_Aug\_Message}(m, m')$ action, where m' is a message added by the CSA for communication with other CSAs. The network may duplicate, lose, and reorder messages arbitrarily (but not corrupt their contents). A message is received in a $\mathit{Receive\_Aug\_Message}(m, m')$ action, which is taken by the network. For each $\mathit{Receive\_Aug\_Message}(m, m')$ action, the CSA at the receiving processor "strips" m' off, and outputs $\mathit{Receive\_Message}(m)$ to the send module. The contents of the m' field of messages is the sole way communication between different CSA is realized.

We assume that when a message is received, lower and upper bounds on its time of transit (which may be 0 and ∞, respectively) are available to the CSA, as functions of the message contents (e.g., its length) and the system specification. The system is defined so that all events are local, i.e., each event is an action of exactly one processor.

In the remainder of this section we define formally specific automata for links and send modules, and give certain conditions that any clock synchronization algorithm must meet.

3.1.1 Send Automaton

Intuitively, the role of a send automaton A_v at processor v is to determine when to send messages and to which neighbor. In general, these decisions may be based (perhaps nondeterministically) on the local history and/or the local time (e.g., timeouts). In this thesis, we concentrate on the highly unstructured automaton, in which messages may be sent at any time to any neighbor.

We assume that send modules have bounded-drift clocks (cf. Def. 2.5). In Figure 3-2 we give a formal specification of a send module. The definition uses the following notation. For each processor v, $\mathcal{N}(v)$ denotes the set of neighbors of v in the underlying graph; Σ denotes a (possibly infinite) *message alphabet*. In Figure 3-2, as we do in the rest of this thesis, we follow the convention that the actions are subscripted by processor names. As we shall see, this is possible since every action in the system is associated with exactly one processor. We usually omit subscripts when the context is clear.

Remark. The basic action of a send module is a point-to-point send. Our definition of send modules includes all possible behaviors of message sends. In particular, a broadcast or a multicast of a message to many processors can be modeled by many send actions taken

Sites: a single site v

State:

now: a non-negative real number, initially 0
local_time: a real number, initially arbitrary

Actions:

$Receive\_Message_v^u(m)$, *for* $m \in \Sigma$ *and* $u \in \mathcal{N}(v)$ *(input)*
Pre: *none*
Eff: *none*

$Send\_Message_v^u(m)$, *for* $m \in \Sigma$ *and* $u \in \mathcal{N}(v)$ *(output)*
Pre: *none*
Eff: *none*

ν: *(time passage)*
Pre: $b > 0$
$\underline{\varrho} \leq r \leq \overline{\varrho}$
Eff: $now \leftarrow now + b$
$local\_time \leftarrow local\_time + r \cdot b$

Figure 3-2: *Specification of a send module A_v at site v with a $(\underline{\varrho}, \overline{\varrho})$-clock*

Sites: *none*

State

now: *non-negative real number, initially* 0
Q: *a multiset of triples* $(m_1, m_2, t) \in \Sigma \times \Sigma' \times \mathbf{R}^+$, *initially* $\emptyset$

Transitions

$Send\_Aug\_Message_u^v(m_1, m_2)$, *where* $m_1 \in \Sigma$, $m_2 \in \Sigma'$ *(input)*
Eff: *choose an arbitrary integer* $i \geq 0$
do i times
put (m_1, m_2, t) *in* Q, *where* t *is an arbitrary number in* $[L(m_1), H(m_1)]$

$Receive\_Aug\_Message_u^v(m_1, m_2)$, *where* $m_1 \in \Sigma$, $m_2 \in \Sigma'$ *(output)*
Pre: $(m_1, m_2, 0) \in Q$
Eff: *remove a triple* $(m_1, m_2, 0)$ *from* Q

ν : *(time passage)*
Pre: $0 \leq b \leq t$ *for all* $(m_1, m_2, t) \in Q$
Eff: $Q \leftarrow \{(m_1, m_2, t - b) \mid (m_1, m_2, t) \in Q\}$
$now \leftarrow now + b$

Figure 3-3: *Specification of a link automaton* L_{vu}.

at the same real time. Notice also that a send automaton may stop sending messages at some point, thus behaving like a process that crashed.

Example. Consider once again the SENDER automaton defined in Figure 2-2. It has the same action signature as the general send module of Figure 3-2, but it is slightly more structured: the *Send_Message* action is not always enabled in SENDER. It is therefore clear that the set of timed traces of SENDER is a strict subset of the set of timed traces of the general send automaton of Figure 3-2. ∎

3.1.2 Network

The network is modeled as a collection of links which facilitate communication among the processors. Each link from a processor v to a processor u has $Send\_Aug\_Message_v^u$ input action (generated by processor v), and $Receive\_Aug\_Message_u^v$ output action, (generated at processor u).[1] We assume very little about the faithfulness of the links: messages may be

[1]The interface between links and processors is sketched in Figure 3-1; a formal description is given in Section 3.1.4, after we define the CSA modules in Section 3.1.3.

lost, duplicated, or re-ordered. We only require that any message received was indeed sent (i.e., no corruption of message contents). We also require that the transmission time of each message received is within some (possibly infinite) interval which is known at the receive point.

More precisely, we associate with each directed link (v, u) a *link automaton* L_{vu} which is responsible for the delivery of messages from v to u. The messages have the form (m_1, m_2), where $m_1 \in \Sigma$ and $m_2 \in \Sigma'$, for some message alphabeta Σ and Σ'. L_{vu} has no sites (i.e., no local clocks), but it satisfies the following timing specification. For any *Receive_Aug_Message*(m_1, m_2) step of the system we assume the existence of two numbers $0 \leq L(m_1) \leq H(m_1) \leq \infty$, such that if the receive event occurs at real time t, then the (unique) send event of this message must have occurred within the time interval $[t - H(m_1), t - L(m_1)]$. The number $L(m_1)$ is called the *latency lower bound* of m_1, and $H(m_1)$ is called the *latency upper bound* of m_1. Note that the latency bounds for a message (m_1, m_2) may depend only on m_1.

A complete description of a L_{vu}-automaton is is given in Figure 3-3.

Remarks.

1. In the formal description of Figure 3-3, latency bounds are determined when a message is *input* into the link. This is done for convenience only. In an equivalent formalization, the latency bounds are determined only when a message is *output*. (The latter formulation may seem more realistic in the sense that transmission time can be better estimated upon delivery than upon sending.) The fact that we shall use in the sequel is that when a message is received, one can determine, from the system specifications and the contents of the message, what are the latency time bounds for that message.

2. Note that the specification of the link is very general. In particular, a link may stop delivering messages starting from some point, thus behaving like a crashed link. However, the link specification guarantees that if a message is received, then it was sent, i.e., there is no corruption of messages.

Example. Let us define a particular kind of links we call *perfect asynchronous links*. For these links, the sequence of messages received is exactly the sequence of messages sent, i.e., message are never lost, created, duplicated, nor re-ordered. The timing specification of these links, however, is the loosest possible: the latency bounds are 0 (lower bound)

Sites: none

State

now: non-negative real number, initially 0
Q: a queue of triples $(m_1, m_2) \in \Sigma \times \Sigma'$, *initially empty*

Transitions

Send_Aug_Message$_u^v(m_1, m_2)$, *where* $m_1 \in \Sigma$, $m_2 \in \Sigma'$ *(input)*
Eff: *enqueue* (m_1, m_2) *in* Q

Receive_Aug_Message$_u^v(m_1, m_2)$, *where* $m_1 \in \Sigma$, $m_2 \in \Sigma'$ *(output)*
Pre: (m_1, m_2) *is in the head of* Q
Eff: *remove head of* Q

ν : *(time passage)*
Pre: $b \geq 0$
Eff: $now \leftarrow now + b$

Figure 3-4: *Specification of a perfect asynchronous link from* v *to* u.

and ∞ (upper bound) for all messages (see formal description in Figure 3-4). A perfect asynchronous link is just a special case of the general link of Figure 3-3, in the sense that the set of timed traces of a perfect asynchronous link is a subset of the set of timed traces of general links.

3.1.3 Clock Synchronization Algorithm (CSA)

The CSA uses the readings of the local clock, and the messages sent and received, in order to carry out some synchronization task (the definition of particular tasks is deferred to later chapters). In this subsection we specify requirements that must be met by any CSA, and point out what remains unspecified.

Interface

CSA modules use two message alphabets for communication, Σ and Σ', where Σ is used by the send automaton, and $\Sigma \times \Sigma'$ is used by the links. The CSA module at processor v has the action signature described in Figure 3-5.

For output, CSA modules may have additional variables or actions. The definitions depend on the specific synchronization task considered, which in turn depend the on definition

Input actions

Send_Message$_v^u(m)$, *for* $m \in \Sigma$ *and* $u \in \mathcal{N}(v)$.
Receive_Aug_Message$_v^u(m_1, m_2)$ *for* $(m_1, m_2) \in \Sigma \times \Sigma'$ *and* $u \in \mathcal{N}(v)$.

Output actions

Send_Aug_Message$_v^u(m_1, m_2)$ *for* $(m_1, m_2) \in \Sigma \times \Sigma'$ *and* $u \in \mathcal{N}(v)$.
Receive_Message$_v^u(m)$, *for* $m \in \Sigma$ *and* $u \in \mathcal{N}(v)$.

Figure 3-5: *Interface of a CSA at processor v*

of the full clock synchronization systems. We therefore defer them to Section 4.1.

Non-Interfering Filtering

The CSA modules use piggybacking on the messages generated by the send modules in order to communicate among themselves. A CSA is not allowed to interfere with message traffic by delaying messages or by deleting parts of their contents. Informally, we think of the CSA as a filter that relays incoming and outgoing messages instantaneously between the send and the link modules (see Figure 3-1), while "sticking" a few extra bytes on each outgoing message, and "stripping" the corresponding bytes from incoming messages. We call this property *non-interfering filtering.*

To capture this property formally, we define an auxiliary notion of a *generic CSA* in Figure 3-6. There, time passage is blocked when there is some message to be processed by the CSA. Using the specification of the generic CSA, we define non-interfering filtering.

Definition 3.1 *A CSA is said to have the* non-interfering filtering *property if its set of timed traces is a subset of the set of timed traces of the generic CSA of Figure 3-6.*

Remark. Notice that in an execution of an automaton with the non-interfering filtering property, there is a natural correspondence between the *Receive_Message* and the *Receive_Aug_Message* events, and between the *Send_Message* and the *Send_Aug_Message* events.

Sites: *a single site* v

State

now: *non-negative real number, initially* 0
$local\_time_v$: *real number, initially arbitrary*
Q_i: *queue for symbols of* Σ, *initially* $\emptyset$
Q_o: *queue for symbols of* $\Sigma \times \Sigma'$, *initially* $\emptyset$
active: *Boolean flag, initially* FALSE

Actions

$Send\_Message_v^u(m)$ *(input)*
Eff: *enqueue* m *in* Q_o
$active \leftarrow$ TRUE

$Send\_Aug\_Message_v^u(m_1, m_2)$ *(output)*
Pre: m_1 *is at the head of* Q_o
Eff: *remove head of* Q_o
if $Q_o = Q_i = \emptyset$ **then** $active \leftarrow$ FALSE

$Receive\_Aug\_Message_v^u(m_1, m_2)$ *(input)*
Eff: *enqueue* m_1 *in* Q_i
$active \leftarrow$ TRUE

$Receive\_Message_v^u(m_1)$ *(output)*
Pre: m_1 *is at the head of* Q_i
Eff: *remove head of* Q_i
if $Q_o = Q_i = \emptyset$ **then** $active \leftarrow$ FALSE

ν : *(time passage)*
Pre: $active =$ FALSE
$b > 0$
$\underline{\varrho} \leq r \leq \overline{\varrho}$
Eff: $now \leftarrow now + b$
$local\_time \leftarrow local\_time + r \cdot b$

Figure 3-6: *Code for a generic CSA with* $(\underline{\varrho}, \overline{\varrho})$*-clock.*

Admissible CSAs

We now define formally the requirements of clock synchronization algorithms. In addition to formalizing our requirement that CSAs are allowed to use only piggybacking for communications, we impose a couple of additional technical requirements; these rule out algorithms which are possible in our formal model, but are usually infeasible in practice.

First, we rule out the possibility that a CSA senses time passage directly: time passage is confined to affect directly only the local clocks, and the CSAs are affected only by changes in the local clocks. This requirement is formalized by the concept of *real-time blindness* (cf. Definition 2.6). Recall that the state of a real-time blind automaton can be decomposed to real time, local times, and basic components. We remark that unless a CSA is trivial, its output is defined in terms of its basic state.

Secondly, notice that in our model, the initial state provides an artificial synchronization point for all processors in the system. Specifically, it is possible that upon initialization, all CSA modules will record the initial value of their local time, thereby getting an accurate snapshot of the local clocks in a perfectly synchronized manner. We rule out such algorithms since the synchronous initialization point is only a convenient abstraction, and cannot usually be implemented in practice. Formally, we require all start states of a CSA automaton to be *quiescent* (see Definition 2.7 for details). Intuitively, the implication of having a quiescent initial state is that the automaton cannot "tell" how much time has elapsed since the (abstract) initialization until the first local input action. Technically, no locally-controlled actions are enabled at a quiescent state: only time passage and input actions are enabled. Formally, we have the following lemma.

Lemma 3.1 *Let $e = \langle \omega_0 \pi_1 \omega_1 \ldots \rangle$ be an execution fragment of an automaton A. If for some i and t we have that the state $\omega_i(t)$ is quiescent, then the action π_{i+1} (if it exists) is an input action.*

Proof: If π_{i+1} does not exist, there is nothing to prove. Otherwise, we have that either $\omega_i(t) = l\_state(\omega_i)$ or else $\omega_i(t) \xrightarrow{\nu} l\_state(\omega_i)$. In both cases, by Definition 2.7, it must be the case that $l\_state(\omega_i)$ is quiet, i.e., only time passage and input actions are enabled in $l\_state(\omega_i)$. Since e is an execution fragment, π_{i+1} is enabled in $l\_state(\omega_i)$ and $\pi_i \neq \nu$, and the lemma follows. ∎

We summarize formally all the requirements a CSA has to satisfy in the following definition.

Definition 3.2 *A mixed automaton is called an* admissible CSA *if it has the external interface specified in Figure 3-5, it has the non-interfering filtering property as specified by Definition 3.1, it is real-time blind as specified in Definition 2.6, and all its initial states are quiescent as in Definition 2.7.*

Henceforth, we restrict our attention to admissible CSAs only.

Latitude in CSA Specification

Definition 3.2 imposes a few severe limitations on CSAs. Let us explain roughly what remains to be defined in a particular implementation of a CSA. First, the definition of an admissible CSA does not specify how to compute the output. Secondly, by the non-interfering filtering property, whenever a $Send\_Message(m_1)$ occurs, a CSA must output a $Send\_Aug\_Message(m_1, m_2)$ action, but m_2 is not specified.

The intuition is that CSA modules have to produce some output (which may be either some values, or some special action). To this end, CSA modules may have additional basic state components, and they can communicate among themselves by using the "m_2" field of the messages.

3.1.4 Clock Synchronization Systems

Having defined the individual components, we are now in a position to define the concept of clock synchronization system. A clock synchronization system is defined by the composition of a collection of send automata, link automata, and CSA automata. Formally, we first compose pairs of send automata and CSAs that share a site. As mentioned before, we call the resulting single-site mixed automaton a *processor*. We require that for each site there is exactly one send module and one CSA (see Figure 3-1). To create the system automaton, we compose the processors with the link automata.

In our definition of systems, each non time-passage action has a naturally associated site of occurrence (there are no internal actions of the link automata). We use this association to define the *local time of occurrence* for each step in an execution. E.g., the local time of occurrence of a $Send\_Message_v^u(m)$ step in a given execution is $local\_time_v(Send\_Message_v^u(m))$.

A clock synchronization system (excluding the CSAs) is thus specified up to clock drift bounds and message latency bounds. We shall refer to these as the *real-time specification* of the system (a formal definition is given later). We assume that the real-time specification of the system can be used by the CSA modules. In other words, the code for a CSA can refer to clock drift bounds and message latency bounds. We argue that this assumption is reasonable. For clocks, one usually has some bounds provided by the manufacturer. For messages, some universal latency bounds are always valid: in all physical systems, the transmission time of any message is at least 0 and at most ∞. In many cases sharper bounds are known. As we shall see, even using the universal bounds some non-trivial synchronization can be attained by the CSAs. Sharpening the bounds may only result in tighter synchronization.

3.1.5 Example: the Simplified Network Time Protocol (SNTP)

In this section we give a concrete example of a clock synchronization system. Our example is based on NTP (Network Time Protocol), the clock synchronization algorithm used over the Internet [26]). We present a simplified version of an NTP system, which we call below SNTP.

In SNTP, we have only two processors, s and v, connected by a bidirectional communication link. Both processors have drift-free clocks. The particular synchronization task we consider is that v needs to bound, at all times, the current reading of the clock of s. (This is a special case of the "external synchronization" task, studied in Chapter 6.) Formally, we require that the CSA module at v maintains two *output variables*, denoted ext_L and ext_U, such that at any state x, $local\_time_s(x) \in [ext\_L, ext\_U]$.

The send and the link automata of SNTP are more structured than the general modules defined in Section 3.1. Specifically, the system architecture is as follows.

The send modules in SNTP are such that periodically, v sends a message to s, which in turn responds by sending a message back to v.[2] The link automata in an SNTP system (L_{sv} and L_{vs}) are perfect asynchronous links (cf. Figure 3-4), i.e., all messages are delivered in order, exactly once with latency bounds 0 (lower bound) and ∞ (upper bound).

Before we describe the way the CSAs work in SNTP, notice that since the clocks of v and

[2]The SENDER automaton of Figure 2-2 can serve as a specification for the send module of v; the send module of s can be specified as a slight variant of SENDER, where the *pend* flag is initially FALSE.

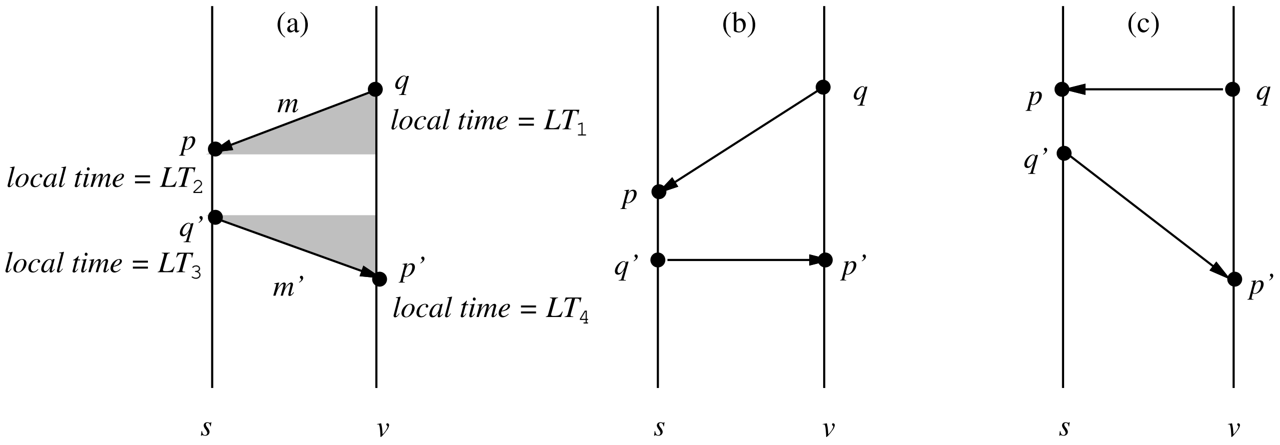

Figure 3-7: *The total transit time of* m *and* m', TT, *is the length of the shaded interval on* v*'s axis in (a). In (b),* m *is in transit* TT *time units, and in (c)* m' *is in transit* TT *time units.*

s are drift-free, the difference between them is the same at all states of a given execution. Therefore, in order to obtain bounds on the local time of s, it is sufficient to have the local time at v, and bounds on the difference between the local time of v and of s at any state.

We now describe the CSA modules of SNTP with the aid of a concrete example (a formal description is given in Figures 3-8 and 3-9). Consider the scenario depicted in Figure 3-7(a), where v sends a message m to s, and s responds by sending m' to v. The CSA modules work as follows. When m is sent by v (point q), the CSA at v records the local time of the send event in the variable LT_1, i.e., it sets $LT_1 = \mathit{local\_time}(q)$. When m is received by the source processor (point p), it records the local time of that event in the variable LT_2, i.e., $LT_2 = \mathit{local\_time}(p)$. When the source sends m' (point q'), m' contains the values of LT_2 and of the local time of the send event, denoted $LT_3 = \mathit{local\_time}(q')$.

When m' is received at v (point p'), v calculates TT, the total transit time of both messages: denoting $LT_4 = \mathit{local\_time}(p')$, this can easily seen to be $TT = (LT_4 - LT_1) - (LT_3 - LT_2)$ (see Figure 3-7 (a)).

Finally, bounds on the difference between v's clock and s's clock are obtained by bounding the local time at the source, at the point at which m' is received at v. The idea is as follows. Let x denote the state of the system immediately after m' is received. Since m' is in transit at least 0 time units (Figure 3-7 (b)), it must be the case that the local time at the source when m' is received at v is at least LT_3, i.e., $\mathit{local\_time}_s(x) \geq LT_3$. On the other hand, since m' was in transit at most TT time units (Figure 3-7 (c)), it must also be the case that the local time at the source when m' is received at v is at most $LT_3 + TT$, i.e., $local\_time_s(x) \leq LT_3 + TT$. Since the local time of v at x is LT_4, and since the difference in local times between v and s is fixed throughout the execution, we have, for any state y in the execution

$$\begin{aligned} local\_time_s(y) - local\_time_v(y) &= local\_time_s(x) - local\_time_v(x) \\ &\in [LT_3 - LT_4\ ,\ LT_3 + TT - LT_4]\ , \end{aligned}$$

and hence,

$$local\_time_s(y) \in [local\_time_v(y) + LT_3 - LT_4\ ,\ local\_time_v(y) + LT_3 + TT - LT_4]\ .$$

When m' is received the local time at v is LT_4, and hence, at that time v sets $ext\_L = LT_3$ and $ext\_U = LT_3 + TT$. Whenever the local time increases at v, the variables ext_L and ext_U are increased by the same amount.

It is easy to verify that the CSAs in SNTP are admissible in the sense of Def. 3.2. First, the CSA modules have the interface of Figure 3-5. Secondly, the CSA modules satisfy the non-interfering filtering property: in fact, their code is based on the code of the generic CSA in Figure 3-6. Thirdly, the CSA modules are easily seen to be real-time blind: their state readily has *now* and *local_time* components, and the rest is the *basic* component. (Notice that the output variables are part of the *basic* component.) It is simple to verify that the transitions depend only on the basic and the local time components of the clock specification. Finally, the initial state of the CSA modules are quiescent, as the only actions enabled at any state reachable from the initial states by time passage are inputs and time passage.

3.2 Environments and Bounds Mapping

In this section we take the final step in modeling clock synchronization systems. We divide the system into two parts, one consists of the CSA modules, and the remainder is called the environment. Intuitively, the idea is to view the aggregate of all send and link automata as a single *environment automaton* (see Figure 3-10), where the goal of the CSA modules is to try to get the tightest possible logical time for each observable behavior of the environment.

Sites: *a single site* v

State

now: *non-negative real number, initially* 0
local_time: *real number, initially arbitrary*
ext_L: *real number, initially* $-\infty$
ext_U: *real number, initially* ∞
Q_i: *queue for symbols of* Σ, *initially* $\emptyset$
Q_o: *queue for symbols of* $\Sigma \times \mathbf{R}^2$, *initially* $\emptyset$
active: *Boolean flag, initially* FALSE
LT_1: *a real number, initially undefined*

Actions

$Send\_Message_v(m)$ *(input)*
Eff: *enqueue* m *in* Q_o
$active \leftarrow$ TRUE
$LT_1 \leftarrow local\_time$

$Send\_Aug\_Message_v(m_1, 0, 0)$ *(output)*
Pre: m_1 *is at the head of* Q_o
Eff: *remove head of* Q_o
if $Q_o = Q_i = \emptyset$ **then** $active \leftarrow$ FALSE

$Receive\_Aug\_Message_v(m_1, \langle LT_2, LT_3 \rangle)$ *(input)*
Eff: *enqueue* m_1 *in* Q_i
$active \leftarrow$ TRUE
$LT_4 \leftarrow local\_time$
$TT \leftarrow (LT_4 - LT_1) - (LT_3 - LT_2)$
$ext\_L \leftarrow LT_3$
$ext\_U \leftarrow LT_3 + TT$

$Receive\_Message_v(m_1)$ *(output)*
Pre: m_1 *is at the head of* Q_i
Eff: *remove head of* Q_i
if $Q_o = Q_i = \emptyset$ **then** $active \leftarrow$ FALSE

ν : *(time passage)*
Pre: $active =$ FALSE
$b > 0$
Eff: $now \leftarrow now + b$
$local\_time \leftarrow local\_time + b$
$ext\_L \leftarrow ext\_L + b$
$ext\_U \leftarrow ext\_U + b$

Figure 3-8: *Code of the CSA module in SNTP for processor* v *(single round-trip).*

Sites: *the source site s*

State

now: non-negative real number, initially 0
local_time: real number, initially arbitrary
Q_i*: queue for symbols of* Σ*, initially* $\emptyset$
Q_o*: queue for symbols of* $\Sigma \times \mathbf{R}^2$*, initially* $\emptyset$
active: Boolean flag, initially FALSE
LT_2*: a real number, initially undefined*

Actions

$Receive\_Aug\_Message_s(m_1, 0, 0)$ *(input)*
Eff: *enqueue* m_1 *in* Q_i
active $\leftarrow$ TRUE
$LT_2 \leftarrow local\_time$

$Receive\_Message_s(m_1)$ *(output)*
Pre: m_1 *is at the head of* Q_i
Eff: *remove head of* Q_i
if $Q_o = Q_i = \emptyset$ **then** *active* $\leftarrow$ FALSE

$Send\_Message_s(m)$ *(input)*
Eff: *enqueue* m *in* Q_o
active $\leftarrow$ TRUE

$Send\_Aug\_Message_s(m_1, LT_2, LT_3)$ *(output)*
Pre: m_1 *is at the head of* Q_o
$LT_3 = local\_time$
Eff: *remove head of* Q_o
if $Q_o = Q_i = \emptyset$ **then** *active* $\leftarrow$ FALSE

ν : *(time passage)*
Pre: *active* = FALSE
$b > 0$
Eff: $now \leftarrow now + b$
$local\_time \leftarrow local\_time + b$

Figure 3-9: *Code of the CSA module in SNTP for processor s.*

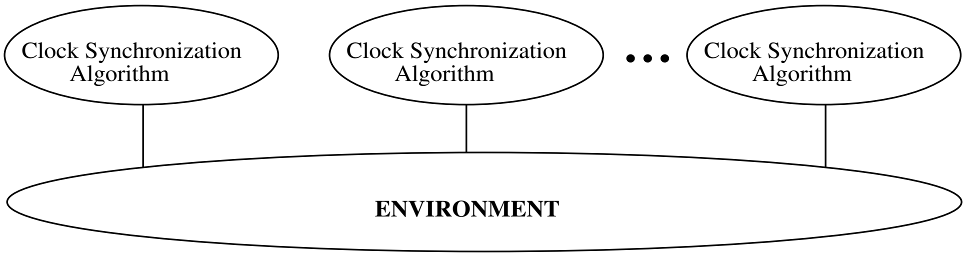

Figure 3-10: *The conceptual arrangement of the automata at a clock synchronization system for the local competitiveness model.*

In Section 3.2.1 we isolate the relevant information in executions of environments in the notions of *pattern* and *views*. A pattern contains all the events with their real and local time of occurrence, while a view does not contain the real time of occurrence. In Section 3.2.2 we define the concept of *local view* at a point in the execution, which is the portion of the view that can be known at that point. In Section 3.2.3 we formalize the real-time specification of a system in the definition of *bounds mapping.* This definition allows us to treat message latency bounds and clock drift bounds in a uniform way. The bounds mapping derived from the real-time specification of the system is called the *standard bounds mapping.*

3.2.1 Environments, Patterns, Views

We start with a formal definition of the notion of environment. Recall that the definition of a send automaton includes the definition of the clock at its site. The environment automaton defined below, therefore, controls the local clocks, message generation, and message delivery in a clock synchronization system.

Definition 3.3 (Environments) *Given a clock synchronization system, the* environment *is the mixed automaton defined by the composition of all send and link automata.*

Our main interest is in executions of environments. The notion of execution contains a great deal of information: for example, at any given time, the state of a link describes precisely, how many copies of each message are in transit and when will they be delivered. For synchronization purposes, however, it seems sufficient to match receive events with send events, ignoring the interim. The concepts of *patterns* and *views* defined below get rid of information in executions which is irrelevant for synchronization. Intuitively, a view contains a set of points (which may be actions or just “placeholders” called *null* points), with a graph structure which describes their order of occurrence, and a local time attribute for each point; a pattern contains also a real-time attribute for each point. The graph structure is essentially the one described by Lamport [16]. Let us recall the following standard graph-theoretic definitions.

Definition 3.4 *Let $G = (V, E)$ be a directed graph. A sequence $p_0, p_1 \ldots, p_k$ is a* path *from p_0 to p_k in G if $p_i \in V$ for $i = 0, 1, \ldots, k$, and $(p_{i-1}, p_i) \in E$ for $i = 1, 2, \ldots, k$. A path from p_0 to p_0 is a* cycle*. A point p is said to be* reachable *from a point q if there is a path from q to p.*

Before we make the definition, recall that in an execution, each event has its *real time of occurrence*; since in clock synchronization systems each event has a unique processor in which it occurs, we also have a unique *local time of occurrence* for each event.

Definition 3.5 (Patterns and Views) *Given an environment automaton A, a* view *is a pair $(G, local\_time)$, where:*

- *$G = (V, E)$ is a directed graph. Each* point *$p \in V$ is either an action of a send automaton in A, or a* null point *that is said to occur at some processor. The* arc *set E is such that for each processor v, the subgraph induced by the set of all points that occur at v is a directed path; in addition, for each $Receive\_Message_v^u(m)$ point in V there is an arc $\left(Send\_Message_u^v(m), Receive\_Message_v^u(m)\right) \in E$.*
- *$local\_time$ is a mapping from the point set V to $\mathbf{R}$. For a point $p \in V$, $local\_time(p)$ is called the* local time of *p.*

A pattern *is a triple $(G, local\_time, now)$, where $(G = (V, E), local\_time)$ is a view, and now maps the points of V to $\mathbf{R}^+$. For a point $p \in V$, $now(p)$ is called the* real time of *p.*

Note that views and patterns contain only actions of the send automata. This information is sufficient, since by the non-interfering filtering property, CSAs must relay messages instantaneously between the send automata and the links. In addition, recall that actions of the links contain the messages "piggybacked" by the CSA modules, and therefore the message contents depend on the specific CSAs in the system. In our definition, the view or the pattern of an execution of an environment automaton is independent of the CSAs.

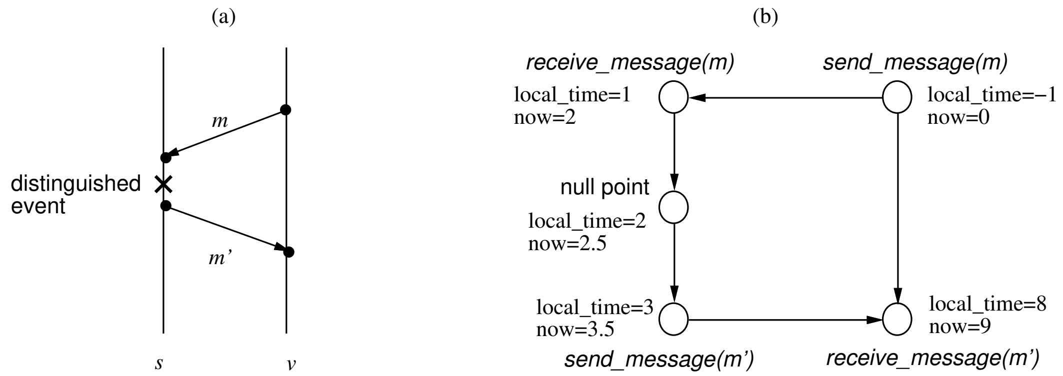

Figure 3-11: *An example of a scenario (a) with its pattern (b). Without the now attributes of the points, the pattern is a view.*

Example. Let us exemplify the concepts of views and patterns using a scenario that was mentioned in the Introduction. We have a system that consists of two processors s and v, connected by a bidirectional communication link. In Figure 3-11 (a) we give a time-space diagram of the following scenario. At real time 0, processor v, whose local clock shows -1, sends a message m to s; processor s receives m at real time 2, when its local clock shows 1. Some distinguished event occurs at s at real time 2.5, when its local clock shows 2. (This event may be an internal event such as flipping a bit, or just the fact that the local clock shows 2.) At real time 3.5, when the local clock of s shows 3, s sends a message m' to v; m' is received at v at real time 9, when its local clock reads 8.

In Figure 3-11 (b) we give an illustration of the pattern based on this scenario, with a null point for the distinguished event. If we remove the *now* attributes of the points in the pattern, the result is a view. ∎

Remarks.

1. Null points in views have only two attributes, namely site of occurrence and local time of occurrence. (In patterns, they also have real time of occurrence.) Null points will be used to enable us to refer to points in which there is no action of the environment.

2. Notice that given an execution of the environment automaton (or a clock synchronization system), its pattern and its view (without null points) are naturally defined, where for each event there is a point, and for each point there is an outgoing arc connecting it to the point that corresponds to the next event that occurs at the same processor (if such a point exists), and each receive point has an incoming arc from the the send point of the corresponding message. Similarly, we can speak about the view of a pattern.

3. The reachability relation in views and patterns of executions is essentially the "happened before" relation described by Lamport [16]: a point p is reachable from a point q in the graph of a view of an executions if and only if q "happened before" p.

Introducing null points into views and patterns. We shall introduce null points into views (and patterns) by stating their processor of occurrence and local time (for patterns, we shall also state their real time). We use the following convention: when introducing into a view $\mathcal{V}$ a null point p_v that occurs at a processor v at local time T_v, the resulting view contains a new point only if there is no other point in $\mathcal{V}$ that occurs at v at local time T_v. In case $\mathcal{V}$ is extended, the modification of the arc set is naturally given: let p_0 be the point that occurs at v with highest local time such that $local\_time(p_0) < T_v$, and let p_1 be the point that occurs at v with smallest local time such that $local\_time(p_0) > T_v$. In the view that contains the null point p_v, we have the additional edges (p_0, p_v) if p_0 exists, and (p_v, p_1) if p_1 exists, and we delete the arc (p_0, p_1) if both p_0 and p_1 exist.

We follow the same procedure when introducing null points into patterns.

3.2.2 Local Views

The motivation for the definition of a view is algorithmic: CSA modules have access only to the information contained in views, as opposed to patterns. (A precise statement of this intuition is formalized in Theorem 3.4.) However, views are defined with respect to a complete execution, while we shall usually require CSA modules to produce output before an (infinite) execution is over... To capture this idea, we define the concept of local view at a point.

Definition 3.6 (Local View) *Given a view $\mathcal{V} = (G, local\_time)$ and a point $p_0 \in \mathcal{V}$, the* local view *of $\mathcal{V}$ at p_0, denoted* $\texttt{prune}(\mathcal{V}, p_0)$*, is the restriction of $\mathcal{V}$ to the points p' such that p_0 is reachable from p' in G. The* local view of $\mathcal{V}$ at processor v at time T *is defined to be* $\texttt{prune}(\mathcal{V}, p_v)$*, where p_v is a null point that occurs at v at local time T.*

For clock synchronization systems, as defined in this chapter, we have the important property that any local view of an execution may actually be the view of the *full* execution. We prove this formally in Theorem 3.2 below.

First, we define a notion of *pruned execution.* Informally, the pruned execution of an automaton A in a clock synchronization system with respect to some point p is the portion of the execution of A that "happened before" p. An additional complication in the definition is due to the fact that in a view, only actions of the send automata are present; the actions of the link and CSA modules are inferred by the the non-interfering filtering property of the CSAs, which matches *Receive_Message* and *Send_Message* events (of send modules and CSAs) with *Receive_Aug_Message* and *Send_Aug_Message* events (of links and CSAs).

Definition 3.7 *Let e be an execution of a clock synchronization system $\mathcal{S}$, and let p be any point in e. The* pruned execution *of an automaton A with respect to p, denoted* $\mathtt{prune}(e|_A, p)$*, is defined as follows.*

- *If A is a send automaton, then $\mathtt{prune}(e|_A, p)$ is the prefix of $e|_A$ up to the last event q such that p is reachable from q in $\mathcal{V}$.*
- *If A is a CSA automaton at a processor v, then $\mathtt{prune}(e|_A, p)$ is the prefix of $e|_A$ up to the event which corresponds to the last event in $\mathtt{prune}(e|_{B_v}, p)$, where B_v is the send module at v.*
- *If A is a link automaton connecting processors u and v, then $\mathtt{prune}(e|_A, p)$ is the prefix of $e|_A$ up to the last event in either $\mathtt{prune}(e|_{C_v}, p)$ or $\mathtt{prune}(e|_{C_u}, p)$, where C_v and C_u are the CSA modules at v and u, respectively.*

Note that if p is an event of A, then the last action in $\mathtt{prune}(e|_A, p)$ is p.

We can now state and prove the property of local views.

Theorem 3.2 *Let $\mathcal{V}$ be a view of an execution e of a clock synchronization system, and let p be any point (possibly a null point) in $\mathcal{V}$. Then there exists an execution e' of the system whose complete view is $\mathtt{prune}(\mathcal{V}, p)$, and such that for each CSA module C_v, $\mathtt{prune}(e|_{C_v}, p) = \mathtt{prune}(e'|_{C_v}, p)$.*

Proof: We start by defining executions for each component of the system separately.

Consider an arbitrary send module A_v. By the specification of send modules, it is clear that $\mathtt{prune}(e|_{C_v}, p)$ can be extended to a full execution e'_{A_v} of A_v with no events other than the ones in $\mathtt{prune}(e|_{C_v}, p)$. Furthermore, this can be done in a way such that $e|_{A_v}$ and e'_{A_v} have the same clock functions (cf. Def. 2.12).

Next, consider a link automaton L_{vu}. Since link automata can drop messages arbitrarily, we have that for any execution $e_{L_{vu}}$ of L_{vu} and for any point q_v, there exists an execution

$e'_{L_{vu}}$, such that $e'_{L_{vu}}$ and $e_{L_{vu}}$ have the same view up to point q_v, and such that in $e'_{L_{vu}}$ there are no *Receive_Aug_Message* events after q_w. We thus get executions of $e'_{L_{vu}}$ for all links L_{vu} whose views agree with $\mathcal{V}$ for all points up to the last point in $\texttt{prune}(e|_{L_{vu}}, p)$.

Using Corollary 2.4.1, we can obtain from the executions e'_{A_v} of all send modules A_v, and from the executions $e'_{L_{vu}}$ of all links L_{vu}, an execution e'_E of the environment, that has view $\texttt{prune}(\mathcal{V}, p)$, and such that e'_E and e have the same clock functions.

Consider now a CSA module C_v at a processor v. We can extend $\texttt{prune}(e|_{C_v}, p)$ to a full execution e'_{C_v} of C_v that has the same clock function as in $e|_{C_v}$, and in which no further input actions are taken. Since all the output actions C_v may take, by the non-interfering filtering property, are in e'_{C_v}, it must be the case that e'_{C_v} has the same view as $\texttt{prune}(e|_{C_v}, p)$.

By construction, the execution e'_A of the environment and the executions e'_{C_v} of the CSA modules C_v agree on the actions and the clock functions of the sites they share. Hence, using Corollary 2.4.1 once again, we can obtain an execution e' of the system, whose view is $\texttt{prune}(\mathcal{V}, p)$. ▮

3.2.3 Representation of Real-Time Specification

Our next step is to give a more convenient representation for the real-time specification of an environment automaton. Recall that we have modeled real-time specifications using clock drift bounds (denoted $\underline{\varrho}$ and $\overline{\varrho}$) and message latency bounds (denoted $L(m)$ and $H(m)$). In this section we state these specifications as bounds on the difference between the real time of occurrence of pairs of points.

We shall make frequent use of the following concepts.

Definition 3.8 (Actual and Virtual Delays) *Let p and q be two points of a given pattern $\mathcal{P}$. The* actual delay *of p relative to q in $\mathcal{P}$, and the* virtual delay *of p relative to q in $\mathcal{P}$, are defined by*[3]

$$\begin{aligned} act\_del_{\mathcal{P}}(p,q) &= now_{\mathcal{P}}(p) - now_{\mathcal{P}}(q) \ , \\ virt\_del_{\mathcal{P}}(p,q) &= local\_time_{\mathcal{P}}(p) - local\_time_{\mathcal{P}}(q) \ . \end{aligned}$$

[3]Throughout this work, we use the following rule when defining a difference of two quantities: $F(x,y) = f(x) - f(y)$, i.e., subtract the second quantity from the first.

The definition of virtual delays extends naturally when we are given only a view.

We also use the following notion.

Definition 3.9 (Adjacent Points) *Two points p, q in a given view $\mathcal{V} = (G, local\_time)$ are called* adjacent points *if there is a directed arc between them in G.*

More intuitively, the above definition (in conjunction with Def. 3.5) says that two points are called adjacent if they occur one after the other in the same processor, or if one is a send event and the other is the corresponding receive event.

Using the above definitions, we define the key concept of bounds mapping.

Definition 3.10 (Bounds Mapping) *A* bounds mapping *for a view $\mathcal{V}$ is a function B that maps every pair p, q of adjacent points in $\mathcal{V}$ to a number such that $-\infty < B(p,q) \leq \infty$. A pattern with view $\mathcal{V}$ is said to* satisfy *B if for all pairs of adjacent points p, q we have $act\_del(p,q) \leq B(p,q)$.*

The general notion of bounds mapping as defined above is not necessarily related to the real-time specification of the environment. The connection is made in the notion of standard bounds mapping, defined as follows.

Definition 3.11 *Let B be a bounds mapping for a view $\mathcal{V}$ of an execution of a clock synchronization system. B is said to be the* standard bounds mapping *for $\mathcal{V}$ if the following holds.*

- *For a message m with send point p, receive point q, and latency bounds $L(m)$ and $H(m)$, we have $B(q,p) = H(m)$ and $B(p,q) = -L(m)$.*
- *Let p be the immediate predecessor of q at a processor with $(\underline{\varrho}, \overline{\varrho})$-clock. Then $B(q,p) = virt\_del(q,p)/\underline{\varrho}$, and $B(p,q) = virt\_del(p,q)/\overline{\varrho}$.*

The following lemma can be thought of as the "soundness" of the standard bounds mapping.

Lemma 3.3 *All patterns of executions of an environment satisfy their standard bounds mapping.*

Proof: By definitions. ▮

Remarks.

1. It is clear from Definition 3.10 that the notion of bounds mapping is in fact more general than the notion of real time specification used so far: using bounds mapping, we can model clocks with drift bounds that are not fixed.

2. The standard bounds mapping has the property of being stated in terms of quantities that are available to the CSA, either as system specification (i.e., $L(m), H(m), \overline{\varrho}, \underline{\varrho}$), or as the local times. Consequently, we may assume without loss of generality that given an environment, the standard bounds mapping can be used in specifying CSA modules.

3.3 The Completeness of the Standard Bounds Mapping

In this section we state and prove the main property of the system we shall use for proving lower bound results. First, we show that if a given pattern has a view of some execution of the system, and if it satisfies the timing specification of the system, then in fact there exists an execution with that pattern. This can thought of as a richness property of the set of executions of the system. In addition, the theorem below says that regardless of the underlying execution, the basic state of CSA modules (which determines the output) depends only on the view of the execution. To this end, we introduce the following definition.

Definition 3.12 *Two executions $e = \omega_0\pi_1\omega_1\ldots$ and $e' = \omega'_0\pi'_1\omega'_1\ldots$ of a CSA are said to be* equivalent *if the following conditions hold.*

(1) For all i, we have $\pi_i = \pi'_i$ and $local\_time(\pi_i) = local\_time(\pi'_i)$.

(2) For all i, for any state s in the range of ω_i and any state s' in the range of ω'_i, we have $basic(s) = basic(s')$.

Condition (1) says that for all i, the ranges of local times in the corresponding trajectories ω_i and ω'_i are the same. Also, recall that by the real time blindness of CSAs, the basic component of the state is constant over a trajectory, and hence Condition (2) above says that for all i, the basic components of the state in the corresponding trajectories ω_i and ω'_i are the same.

The following theorem can also be viewed as a converse to Lemma 3.3. In a sense, we show that the standard bounds mapping is *complete* with respect to a view.

Theorem 3.4 *Let $\mathcal{V}$ be a view of an execution e of a clock synchronization system $\mathcal{S}$, and let B be the standard bounds mapping for $\mathcal{V}$. Let $\mathcal{P}$ be any pattern of the environment automaton with view $\mathcal{V}$. If $\mathcal{P}$ satisfies B, then there exists an execution e' of $\mathcal{S}$ with pattern $\mathcal{P}$. Moreover, for each CSA module C_v, the executions of C_v in e and e' are equivalent.*

Proof: The proof is straightforward, but somewhat tedious. Our strategy to construct e' is as follows. We first construct individual executions for the send modules, the link automata and the CSA modules of $\mathcal{S}$, based on $\mathcal{P}'$ and on e. Then we apply Corollary 2.4.1 and get an execution e' of $\mathcal{S}$ with the required properties. The idea is that pairs of real and local times given in $\mathcal{P}$ can be used – by interpolation – to define complete clock functions for the desired execution e'. With these clock functions, we get executions of the send automata and the CSA module quite easily, since they are real-time blind. For the link automata, some extra work is needed, because their state is affected directly by time passage.

Defining clock functions. We define a function $local\_time'_v : \mathbf{R}^+ \mapsto \mathbf{R}$ for each site $v \in sites(\mathcal{S})$. These functions describe the local times at the sites as a function of real time. (Whereas a clock function is usually defined in terms of an execution, here we first define the clock function and then proceed to construct the execution.) Some values of the clock function are already specified by the pattern; intuitively, our construction simply connects these values by linear interpolation, with (possibly) some special treatment of the first and last segments. Formally, for each site v, we define a local clock function $local\_time'_v(t)$ for all $t \geq 0$ using the given pattern $\mathcal{P}$ and the following rule.

1. If there exists in $\mathcal{P}$ some point p_i that occurs at v with $now(p_i) = t$, we set $local\_time'_v(t)$ to be $local\_time_{\mathcal{P}}(p_i)$.

2. Otherwise, let p_0 be the point in $\mathcal{P}$ with maximal real time such that p_0 occurs at v and $now(p_0) < t$. Let $t_0 = now(p_0)$ and $T_0 = local\_time(p_0)$. If there is no such point, t_0 and T_0 are undefined. Similarly, let p_1 be the pont in $\mathcal{P}$ with minimal real time such that p_1 occurs at v and $now(p_1) > t$. Let $t_1 = now(p_1)$ and $T_1 = local\_time(p_1)$. If there is no such point, t_1 and T_1 are undefined. We distinguish among the following cases.

(a) If both p_0 and p_1 are undefined (i.e., no point occurs at v), we define for all $t \geq 0$,

$$local\_time'_v(t) = c \cdot t \ ,$$

where c is any constant in the range $[\underline{\varrho}_v, \overline{\varrho}_v]$.

(b) If only p_0 is undefined (i.e., t is smaller than the real time of the first point that occurs at v), we define

$$local\_time'_v(t) = T_1 - c' \cdot (t_1 - t) \ ,$$

where c' is any constant in the range $[\underline{\varrho}_v, \overline{\varrho}_v]$.

(c) If only p_1 is undefined (i.e., t is larger than the real time of the last point that occurs at v), we define

$$local\_time'_v(t) = T_0 + c'' \cdot (t - t_0) \ ,$$

where c'' is any constant in the range $[\underline{\varrho}_v, \overline{\varrho}_v]$.

(d) If both p_0 and p_1 are defined (i.e., there are points that occur at v with real time strictly less and strictly more than t), we define

$$local\_time'_v(t) = T_0 + (t - t_0) \cdot \frac{T_1 - T_0}{t_1 - t_0} \ .$$

Notice that $local\_time'_v$ is well defined in case (2d) since $t_0 < t < t_1$. It is straightforward to verify that the local clock functions thus defined are continuous. Also, since $\underline{\varrho} > 0$ and since $\mathcal{P}$ satisfies the standard bounds mapping, we get that the local clock functions are and monotonically increasing. Therefore, $local\_time'_v$ is invertible (at least) on $[T^s_v, \infty]$, where T^s_v is the local time of the first point in $\mathcal{P}$ that occurs at v (if it exists). We denote the inverse function by by $local\_time^{-1}_v$.

This concludes the definition of the local clock functions. Using these functions, we next define executions of the individual components of the system. The idea is to use the original execution e, keep the local times of the points, but "shift" and "stretch" the real times so that they agree with $\mathcal{P}$.

Send modules. We now construct an execution e'_{A_v} of a send module A_v that agrees with $\mathcal{P}$. Most of the work was already done in the definition of the local clocks, since the state of a send module consists merely of local and real times. More specifically, let the subsequence of actions of A_v in $\mathcal{P}$ be $\mathcal{P}_{A_v} = \langle \pi_1^S, \pi_2^S \ldots \rangle$. Since A_v has no internal actions, all its steps are specified by $\mathcal{P}_{A_v}$. To get a complete description of the desired execution $e'_{A_v} = \langle \omega_0^S \pi_1^S \omega_1^S \ldots \rangle$ of A_v, we need only to specify the trajectories ω_i^S. Recall that the state of a send module is a pair $(\mathit{now}, \mathit{local\_time})$ of real and local time. Let $i \geq 0$, and let $\mathit{now}(\pi_i) \leq t \leq \mathit{now}(\pi_{i+1})$, where we define $\mathit{now}(\pi_0^S) = 0$, and if there is no π_{i+1}, we define $\mathit{now}(\pi_{i+1}) = \infty$. Then we define the trajectory ω_i^S by $\omega_i^S(t) = (t, \mathit{local\_time}'_v(t))$.

It is straightforward to see that e'_{A_v} thus constructed is an execution of A_v: we first need to check that ω_i^S is a trajectory for all $i \geq 0$. This is easy, since the only restriction on time passage steps is that they observe the drift bounds, and this is guaranteed by the construction. Since the discrete actions have no effect on the state, all that remains to be verified is that $\omega_0^S(0)$ is a start state, which is true because $\mathit{now}(\omega_0^S(0)) = 0$ by construction.

CSA modules. Consider a CSA module at site v, and let $e|_{CSA} = \langle w_0 \pi_1^C w \ldots \rangle$ be the projection of e on that module. By Lemma 2.3, $e|_{CSA}$ is an execution of the CSA. We now construct another execution $e'_{CSA} = \langle \omega_0^C \pi_1^C \omega_1^C \ldots \rangle$ of the CSA, which agrees with $\mathcal{P}$ on the visible actions. The first step in the construction is to fix the sequence of actions in e'_{CSA} to be the same as in $e|_{CSA}$. To complete the specification of e'_{CSA}, we need to define the trajectories.

It is convenient to first define local and real times for the steps. For the visible steps in e'_{CSA}, we have local and real times already specified by $\mathcal{P}$. For internal steps, the idea is to keep the local times as in e, and to set the real time to be in accordance with the local clock functions defined above. Specifically, let π_i^C be an internal step of the CSA. We abuse notation slightly and denote by $\mathit{local\_time}_{e|_{CSA}}$ local clock function in e at site v. We define

$$\mathit{local\_time}_{e'_{CSA}}(\pi_i^C) = \mathit{local\_time}_{e|_{CSA}}(\pi_i^C) \; .$$

To set the *now* component, we use the inverse of the local clock function as follows:

$$\mathit{now}_{e'_{CSA}}(\pi_i^C) = \mathit{local\_time}_v^{-1}(\mathit{local\_time}_{e|_{CSA}}(\pi_i^C)) \; , \tag{3.1}$$

i.e., the real time of occurrence of an action π^C is given by the unique t such that $\mathit{local\_time}'_v(t)$ is the local time of occurrence of π^C in $e|_{CSA}$ (we shall see later that this number is well defined).

We now define the trajectories ω_i^C in e'_{CSA} for all $i \geq 0$. Again, we use $e|_{CSA}$. More specifically, to define a trajectory ω_i^C in e'_{CSA}, we use the parallel trajectory w_i in $e|_{CSA}$ as follows. Let $t \in [now(\pi_i^C), now(\pi_{i+1}^C)]$ for any $i \geq 0$ (where we define $now(\pi_0^C) = 0$ and if π_{i+1}^C does not exist, we define $now(\pi_{i+1}^C) = \infty$). The trajectory ω_i^C is defined by

$$
\begin{aligned}
now_{e'_{CSA}}(\omega_i^C(t)) &= t \\
\mathit{local\_time}_{e'_{CSA}}(\omega_i^C(t)) &= \mathit{local\_time}'_v(t) \\
\mathit{basic}_{e'_{CSA}}(\omega_i^C(t)) &= \mathit{basic}_{e|_{CSA}}(w_i(t')) \ ,
\end{aligned}
\tag{3.2}
$$

where t' is any number in the domain of w_i.

Let us show that our construction is well defined. First, note that since $e|_{CSA}$ is an execution of a CSA, its initial state must be quiescent, and hence, by Lemma 3.1, π_1^C is not an internal action of the CSA. Therefore, there is a step of the send module in $\mathcal{P}$ whose local time is $\mathit{local\_time}(\pi_1^C)$, which implies that $\mathit{local\_time}_v^{-1}$ is defined over $[\mathit{local\_time}_{e|_{CSA}}(\pi_1^C), \infty]$. This, in turn, implies that Eq. (3.1) is well defined. Finally, note that by real-time blindness, the *basic* component of the state of a CSA is fixed throughout a trajectory, and therefore Eq. (3.2) is not ambiguous.

Next, notice that conditions (1) and (2) in the statement of the theorem are satisfied by the construction. This is true since for all $i \geq 0$, all the states in the range of ω_i^C have the same basic component, which is the same as the basic component of all states in the range of w_i; in addition, for $i \geq 1$, the intervals of local times in ω_i^C and w_i are the same.

We now show that e'_{CSA} is an execution of the given CSA. To show that we use heavily the real-time blindness property. First, we prove that ω_i^C is a trajectory of the CSA for all $i \geq 0$. Let $s_1 = \omega_i^C(t)$ and be $s_2 = \omega_i^C(t')$ be two states, where $t < t'$. Let s_1^* and s_2^* be the states in the corresponding trajectory w_i that satisfy $\mathit{local\_time}(s_1^*) = \mathit{local\_time}(s_1)$ and $\mathit{local\_time}(s_2^*) = \mathit{local\_time}(s_2)$. This is possible since by construction, w_i and ω_i^L agree on the local time in their endpoints, and since the local clock function is continuous. Also by construction, $\mathit{basic}(s_1) = \mathit{basic}(s_1^*)$ and $\mathit{basic}(s_2) = \mathit{basic}(s_2^*)$; moreover, it is easy to see that $\mathit{local\_time}(s_2) - \mathit{local\_time}(s_1) \in [\underline{\varrho}(t'-t), \overline{\varrho}(t'-t)]$ by the assumption that $\mathcal{P}$ satisfies the standard bounds mapping. Since $s_1^* \xrightarrow{\nu} s_2^*$, we get from the real-time blindness of the CSA and that $s_1 \xrightarrow{\nu} s_2$, as required in this case.

Consider now a discrete action π_i^C. Let $s_1 = l\_state(\omega_{i-1}^C)$, $s_2 = f\_state(\omega_i^C)$, $s_1^* = l\_state(w_{i-1})$, and $s_2^* = f\_state(w_i^C)$. By construction we have that s_1 and s_1^* may differ only in their *now* component, and similarly s_2 and s_2^*. From the construction we also have that $now(s_1) = now(s_2)$ and $local\_time(s_1) = local\_time(s_2)$. Since we know that $s_1^* \xrightarrow{\pi_i^C} s_2^*$, we get from real-time blindness that $s_1 \xrightarrow{\pi_i^C} s_2$, as required for this case. This completes the proof that e'_{CSA} is an execution of the CSA.

Link automata. Consider now a link automaton L_{uv}. By the non-interfering filtering property, in e there exist natural bijections between the $Send\_Aug\_Message_u^v$ actions of L_{uv} and the $Send\_Message_u^v$ actions of A_u, and between the $Receive\_Aug\_Message_v^u$ actions of L_{uv} and the $Receive\_Message_v^u$ actions of A_v. Since all the actions of A_u and A_v appear also in $\mathcal{P}$, using these bijections we can define a sequence $\mathcal{P}_{L_{uv}} = \langle \pi_1^L, \pi_2^L \ldots \rangle$ which contains all the actions of L_{uv} that correspond to actions of A_v in $\mathcal{P}$. Notice also that using these bijections, each event in $\mathcal{P}_{L_{uv}}$ inherits a *now* component, and that the causality mapping γ can be extended so that for each *Receive_Aug_Message* event p there is a *Send_Aug_Message* event q satisfying $q = \gamma(p)$. We use these extended notions in the construction below.

Our goal is to construct an execution $e'_{L_{uv}} = \langle \omega_0^L \pi_1^L \omega_1^L \ldots \rangle$ of L_{uv} that agrees with $\mathcal{P}_{L_{uv}}$. Similarly to the case of send modules, L_{uv} has no internal steps, and hence all the steps π_i^L are already specified by $\mathcal{P}_{L_{uv}}$. It remains to specify the trajectories of $e'_{L_{uv}}$. We shall use the following notation.

Notation 3.13 *The contents of the multiset Q_{uv} at state s is denoted $Q(s)$.*

We define $Q((\omega_0^L(0)) = \emptyset$, and $now(\omega_0^L(0)) = 0$. The rest of the construction is done inductively. Suppose that $f\_state(\omega_i^L)$ is defined. For t in the domain of ω_i^L, we define $now(\omega_i^L(t)) = t$, and $Q(\omega_i^L(t))$ is defined by a bijection from $Q(f\_state(\omega_i^L))$ using the following rule:

$$Q(f\_state(\omega_i^L)) \ni (m_1, m_2, t') \quad \longleftrightarrow \quad (m_1, m_2, t' - t + f\_now(\omega_i^L)) \in Q(\omega_i^L(t)) \, . \qquad (3.3)$$

In other words, the third component t' in each triple (m_1, m_2, t') stored in Q_{uv} at the start of ω_i^L is reduced by the amount of time that has elapsed since the start of ω_i^L. To define the start state of trajectories ω_i^L with $i > 0$, we define $Q(f\_state(\omega_i^L))$ as a modification of $Q(l\_state(\omega_{i-1}))$, with the help of the (extended) causality function γ. Specifically, suppose first that $\pi_i^L = Send\_Aug\_Message(m_1, m_2)$. Then we define

$$Q(f\_state(\omega_i^L)) = Q(l\_state(\omega_{i-1})) \cup \left\{(m_1, m_2, act\_del(\pi_j^L, \pi_i^L)) \; : \; \gamma(\pi_j^L) = \pi_i^L\right\} \; . \quad (3.4)$$

In words, Q_{uv} is augmented by one triple for each copy of (m_1, m_2) that will be received in the future, as specified by γ.

If $\pi_i^L = Receive\_Aug\_Message(m_1, m_2)$, we define

$$Q(\omega_i^L(t)) = Q(l\_state(\omega_{i-1})) \setminus \{(m_1, m_2, 0)\} \; . \quad (3.5)$$

In words, one copy of $(m_1, m_2, 0)$ is removed from Q_{uv}. We show below that $(m_1, m_2, 0) \in Q(l\_state(\omega_{i-1}))$ in this case. This concludes the con of $e'_{L_{uv}}$.

We now have to show that $e'_{L_{uv}}$ is an execution of L_{uv}. The key to the proof is a certain invariant; to state it, we introduce another piece of notation.

Notation 3.14 *For a state s in $e'_{L_{uv}}$, $R(s)$ is the set of all Receive_Aug_Message events that occur after state s and such that for all $p \in R(s)$, $\gamma(p)$ occurs before s.*

With this notation, we state the following invariant, parameterized by a state s of $e'_{L_{uv}}$:

> **Invariant $\mathcal{I}(s)$:** There exists a bijection $R(s) \leftrightarrow Q(s)$ that maps each $(m_1, m_2, t) \in Q(s)$ to a step $\pi_k^L \in R(s)$ such that $\pi_k^L = Receive\_Aug\_Message(m_1, m_2)$ and $now(\pi_k^L) - now(s) = t$.

As a preliminary observation, notice that $\mathcal{I}(s)$ implies that for all $(m_1, m_2, t) \in Q(s)$ we have $t \geq 0$, which implies that $s \in states(L_{uv})$.

Our first step is to prove that if $\mathcal{I}(f\_state(\omega_i^L))$ holds for some $i \geq 0$, then ω_i^L is a trajectory for L_{uv}. Consider two states $s = \omega_i^L(t)$ and $s' = \omega_i^L(t')$ where $t < t'$, and suppose $\mathcal{I}(s)$ holds. We argue that for all $(m_1, m_2, t) \in Q(s)$, we have that $t \geq now(s') - now(s)$: for suppose not, i.e., there exists a triple $M = (m_1, m_2, t)$ with $t < now(s') - now(s)$. Then by $\mathcal{I}(s)$, the corresponding $Receive\_Aug\_Message(m_1, m_2)$ event π_j^L occurs after s, and for that event we have $now(\pi_j^L) = now(s) + t < now(s')$. It follows that $now(s) \leq now(\pi_j^L) < now(s')$, contradicting the assumption that s and s' are states on the same trajectory, i.e., that there is no discrete action that occurs between them. Using this fact, it is easy to verify that $(s, \nu, s') \in trans(L_{uv})$ according to the construction above.

Next, we show that if $\mathcal{I}(s)$ holds, and $s \xrightarrow{\nu} s'$, then $\mathcal{I}(s')$ holds. Let h be the bijection between $R(s)$ and $Q(s)$ that satisfies the requirement of $\mathcal{I}(s)$. Let g be the bijection induced by the construction between the elements of $Q(s)$ and $Q(s')$. More specifically, g is the bijection defined in Eq. (3.3). We thus define h' to be the composition of h and g. It is straightforward to verify that h' satisfies the requirements of $\mathcal{I}(s')$.

We have proven that if $\mathcal{I}(f\_state(\omega_i^L))$ holds, then $\mathcal{I}(\omega_i^L(t))$ holds for all t for which $\omega_i^L(t)$ is defined, and in particular, $\mathcal{I}(l\_state(\omega_i^L))$ holds, if it exists. We now show, by induction on i, that $\mathcal{I}(f\_state(\omega_i^L))$ holds for all $i \geq 0$. Trivially, $\mathcal{I}(f\_state(\omega_0^L))$ holds because $Q(\omega_0^L(0)) = \emptyset$. For the inductive step, let $i > 0$. By the previous claim and the induction hypothesis, $\mathcal{I}(s)$ holds for $s = l\_state(\omega_{i-1}^L)$. Let h denote the bijection that satisfies $\mathcal{I}(s)$. Let $s' = f\_state(\omega_i^L)$. To show that $\mathcal{I}(s')$ holds, we define a bijection h' for s'.

Suppose first that $\pi_i^L = Send\_Aug\_Message_u^v(m_1, m_2)$. Then by construction $Q(s') \supseteq Q(s)$. Furthermore, by Eq. (3.4), there exists a bijection f between $Q(s') \setminus Q(s)$ and $R(s') \setminus R(s)$. We can therefore define h' to be the extension of h by f, and $\mathcal{I}(s')$ in this case.

Suppose now that $\pi_i^L = Receive\_Aug\_Message_v^u(m_1, m_2)$. Notice that by the definition of $R(s)$, we have $\pi_i^L \in R(s)$. Also, by $\mathcal{I}(s)$, we have $M = (m_1, m_2, 0) \in Q(s)$. Moreover, it must be the case that $h(M) = \pi_i^L$. By Eq. (3.5), we have that $Q(s') = Q(s) \setminus \{M\}$, and by definition, we have that $R(s') = R(s) \setminus \{\pi_i^L\}$. We can therefore define h' to be the restriction of h on $Q(s')$ and $R(s')$, and h' satisfies the requirements of $\mathcal{I}(s')$. This completes the inductive step.

Finally, note that the fact that $\mathcal{I}(l\_state(\omega_i^L))$ holds for all $i \geq 0$ implies that by construction,

$$(l\_state(\omega_i^L), \pi_{i+1}^L, f\_state(\omega_{i+1}^L)) \in trans(L_{uv}) \; .$$

We conclude the argument that $e'_{L_{uv}}$ is an execution of L_{uv} by observing the trivial fact that $\omega_0^L(0)$ is a start state of L_{uv}.

Concluding argument. To conclude the proof of the theorem, we argue that there exists an execution e' of $\mathcal{S}$ such that its projections on the send automata, link automata, and CSA automata are the executions constructed above. To do that, we first extend $\mathcal{P}$ to be a form for $\mathcal{S}$. This is straightforward: we insert into $\mathcal{P}$ all visible actions of the sub-executions we constructed, and for each point, we extend the local time to be a times form using the local clock functions. Also, we define a form for $\mathcal{S}$ with start real time $t_s = 0$ and finish real time $t_f = \infty$; for all $v \in sites(\mathcal{S})$ we define local start times $\overline{\boldsymbol{T}}_s(v) = local\_time'_v(0)$, and local finish times $\overline{\boldsymbol{T}}_f(v) = \infty$. Now, to apply Corollary 2.4.1 all that remains is to verify that the local times in the sub-executions constructed above agree on shared sites; but this is immediate, since for each site we used the same local clock function. Therefore, there exists the desired execution e'. ∎

Summary

In this chapter we defined clock synchronization systems, using the mixed I/O automata formalism. Our model is geared towards the local competitiveness analysis presented in Chapter 4. Intuitively, the basic assumptions of the model are as follows.

- The system has an underlying communication graph over which messages are communicated.
- Each processor has a local clock with known bounds on the rate of progress, called *clock drift bounds*.
- When a message is received, there are known bounds on its time of transit, called *message latency bounds*. However, messages may be lost, duplicated, and delivered arbitrarily out of order.
- Send events are generated arbitrarily by a *send module* at each processor.
- The *clock synchronization algorithm* at each processor, abbreviated *CSA*, may only append information to outgoing messages, and strip the corresponding information that arrives on incoming messages. CSAs may not interfere with message traffic otherwise, and their only access to time is via the local clocks.

We also defined the following concepts.

- An *environment* is the composition of all send modules and communication links. Thus an environment controls send and receive events.
- A *pattern* of an execution of an environment is a directed graph that describes the execution, where each event is a *point*, and for each point we have local and real time of occurrence.
- A *view* is a pattern without the real time attribute for points. Views of executions of environments contain information that can be used by CSAs for computation, while the real time information in patterns is available only for analysis.
- a *local view* at a point p is the restriction of the view to all the points that "happened before" p (as defined by Lamport [16]). We proved that any local view of an execution may be the view of a full execution of the system.

- The *virtual delay* of a pair of points, denoted *virt_del*, is the local time of occurrence of the first point minus the local time of occurrence of the second point.

- The *actual delay* of a pair of points, denoted *act_del*, is the real time of occurrence of the first point minus the real time of occurrence of the second point.

- Two points are called *adjacent* if either they occur at the same processor one after the other, or they correspond to the send and receive event of the same message.

- A *bounds mapping* for a view specifies time upper bounds for the actual delays of adjacent points. Bounds mapping describes lower bounds as well, by reversing the order of the points.

- The *standard bounds mapping* is the "official" bounds mapping, derived from message latency bounds, clock drift bounds, and local times.

We also proved the fundamental theorem of our model, which says that all the patterns with a given view which satisfy the standard bounds mapping, are possible patterns of executions of the system. The theorem also implies that the output of CSAs depends only on the view of the execution.

Chapter 4

Problem Statements and Quality Evaluation

In this chapter we define the synchronization tasks considered in this thesis, and the way we evaluate the performance of synchronization algorithms. As we shall see, there is a natural concept of *tightness* of synchronization for the clock synchronization problems we define; the tightness is measured in non-negative real numbers, and an output will be considered "good" if its tightness is small. However, it is not clear *a priori* what is the input for synchronization algorithms. One classical answer for this question is that the input is the system specification. A typical example for this approach is the paper by Halpern *et al.* [13], where designing a synchronization algorithm is viewed as a "game against nature:" an algorithm is called optimal if it produces the best output under the worst-case scenario allowable by the system specification. This approach has the appealing property of robustness, but it may give rise to algorithms that produce the best worst-case result always, even if the actual execution does not happen to be the worst possible (the algorithm given in [13] has this property). This is a disadvantage if the environment is not necessarily adversarial, as may be the case for clock synchronization systems.

Another approach, developed by Attiya *et al.* [3], is that the input for a synchronization algorithm is not only the system specification, but also the actual execution, or more precisely, the *view* of the execution.[1] In this approach, an algorithm is called optimal if it

[1] Recall that views consist of the events and their local times of occurrence, while executions contain also the real times of occurrence, which is not available for computation (see Def. 3.5). We remark that Attiya *et al.* used the term *execution* to denote the concept we call *view*.

produces the best possible output for each given input, i.e., for each possible view of an execution (and the system specification for that view). The latter approach is more attractive since an optimal algorithm in this sense has a stronger guarantee of output quality than the guarantee made by an optimal algorithm in the former sense.

Both approaches of [3] and [13], however, suffer from an important disadvantage, which is that the algorithms they consider are centralized and off-line. More specifically, the algorithms are based on the implicit assumption that all input has been gathered and it is available at a single processor for computation. This is clearly a drawback, since the output of clock synchronization algorithms typically needs to be available all the time, i.e., on-line. For example, in the approach of [3], the input is a view of the execution, which contain certain messages. Notice that this view can be made available at a single processor only if more messages are sent, in which case the view is necessarily extended. Thus an output considered optimal for a view may not be optimal when that view is extended to enable computation.

The approach we present in this chapter can be viewed as a combination of the optimality notion of [3] with the well-known concept of competitive analysis of on-line algorithms [32, 23], using Lamport's causality relation [16]. More specifically, in competitive analysis the quality of the output produced by an on-line algorithm is evaluated at each point with respect to the input known at that point. In the centralized on-line setting, all past input is known, and the future input is unknown. In the distributed setting, even past input is unknown if it is remote and has not been communicated. We therefore define the input at a point to consist of what can be known locally (called *local view* in Def. 3.6). We measure the quality of the output of an algorithm A with respect to the quality of the best possible output for the given local view. We call the ratio between these quantities the *local competitiveness* of algorithm A.

The remainder of this chapter is organized as follows. In Section 4.1 we give formal definitions for the synchronization tasks considered in this thesis. The definition of locally competitive algorithms is given in Section 4.2. In Section 4.3 we discuss the concept of local competitiveness in a more general setting.

4.1 Synchronization Tasks

In this section we define the specific synchronization tasks we consider in this thesis, namely external and internal synchronization. For each problem we give a refined specification of the system architecture, a correctness requirement, and a definition of tightness.

4.1.1 Definition of External Synchronization

The motivation for external clock synchronization is systems where one of the clocks is assumed to show the standard time, and the goal is that all clocks in the system will show this standard time as accurately as possible. The name "external synchronization" stems from the assumption that the designated clock serves as a source of the external standard time into the system. Formally, we shall use the following definition.

An *external synchronization system* is a clock synchronization system with the following properties. There exists a distinguished processor s, called the *source processor*, whose local clock is drift-free. A CSA module at each processor v has two *output variables*, denoted ext_L_v and ext_U_v.

For any given state x, let $source\_time(x)$ denote the local time at the source in x. The *correctness requirement* of an *external CSA* at any processor v is that at every reachable state x, the output variables at v satisfy $source\_time(x) \in [ext\_L_v, ext\_U_v]$.

The *external tightness* of synchronization at processor v at some state is the difference $(ext\_U_v - ext\_L_v)$ at that state.

Remark. An alternative formulation of the problem would be to require the CSAs to produce one number T as an estimate of the current source time, and another number ε that bounds the current difference between the estimate and the source time. While the two specifications are equivalent if ext_L and ext_U are both finite or both infinite, we prefer the (ext_L, ext_U) formulation, since it is slightly more refined: in the case where exactly one of the numbers ext_L or ext_U is finite, the output according to the (T, ε) formulation is the same as for the case where both ext_L and ext_U are infinite.

4.1.2 Definition of Internal Synchronization

We use a variant of the elegant definition of Dolev *et al.* [7] and Halpern *et al.* [13], which we formulate as follows. (A discussion of the definition is given in Chapter 7.)

An *internal synchronization system* is a clock synchronization system, such that each CSA module has a special internal action called $fire_v$, where v is the site of the module.

The *correctness requirement* of the internal synchronization task is that first, each processor v takes a $fire_v$ action exactly once during an execution of the system. And secondly, the CSA at each processor maintains output variables called int_L_v and int_U_v, such that at all states, the real time interval $[now(fire_v) + int\_L_v, now(fire_v) + int\_U_v]$ contains all the *fire* events in the execution. Intuitively, the output variables provide local guarantees for the tightness in which all *fire* actions are produced in the system. Initially, we will have $int\_L = -\infty$ and $int\_U = \infty$, and during the execution, int_L may get larger and int_U may get smaller.

The *internal tightness* at processor v in some state is the difference $(int\_U_v - int\_L_v)$ at that state. The *internal tightness of an execution* at a processor v is the infimum of the internal tightness at v, over all states of the execution. The internal tightness of v in an execution e is denoted $tightness_v(e)$.

4.2 Local Competitiveness

Local competitiveness is our measure of quality of synchronization algorithms. Intuitively, an algorithm is said to be locally K-competitive if its output at any point is at most K times worse than the best possible for the local view at that point. We formalize this intuition for CSAs as follows.

Fix a synchronization problem. As described in Section 4.1, each problem has a predicate that classifies CSAs as "correct" and "incorrect." More specifically, the correctness predicate classifies executions as correct and incorrect; a CSA is correct if all its executions are correct.

In Section 4.1 we also defined, for each synchronization problem, a function called *tightness*, that maps states of CSAs to $\mathbf{R}^+ \cup \{\infty\}$. By real-time blindness, the tightness is a function only of the *basic* component of the state. Recall that by Theorem 3.4, the basic component of a state of a CSA module in an execution depends only on the view of the execution. Hence, given a CSA module (in either an internal or an external synchronization system), the *tightness of the view* at a given point is well defined. (If the CSA is not deterministic, then the tightness is a non-deterministic function of the local view.) Using the notions of correct CSAs and tightness of views, we define the key concept of local competitiveness as follows.

Definition 4.1 *Let $\mathcal{A}$ be the set of all correct CSAs for a given environment. Let $\Theta_{A,v}(\mathcal{V},T)$ be the tightness of synchronization in executions with local view $\mathcal{V}$ of a processor v at local time T, for a system with a CSA module $A \in \mathcal{A}$. An algorithm A is said to be* locally K-competitive *if for all views $\mathcal{V}$, processors v and local times T,*

$$\Theta_{A,v}(\mathcal{V},T) \le K \cdot \inf\left\{\Theta_{A_0,v}(\mathcal{V},T) : A_0 \in \mathcal{A}\right\} .$$

The least number K such that A is K-competitive is the local competitive factor *of A. A locally 1-competitive algorithm is also called* optimal.

Remarks.

1. Recall that our model definitions allow for nondeterministic CSAs, i.e., CSAs whose output is not a deterministic function of the view. In this case, the correctness requirement is that all possible executions are correct. On the other hand, we can define the tightness of a view to be the least tightness over all executions with the given view, which means that we consider the best possible choices made at the non-deterministic choice points, so long as they produce correct results.

2. It is important to notice that in principle, there always exists a *full information protocol* which is optimal: in this algorithm, the processors send their complete view in every message; how to determine the output depends on the specific problem being solved, but clearly optimal output can be computed since all the relevant information is available locally at each processor, simply because all *possible* information is there! It is also clear, however, that the full information protocol is usually not practical. From the communication perspective, the message size blows up rapidly to fantastic lengths; and from the processing perspective, it may well be the case that extracting the output from the "full information" is computationally infeasible. The goal of the designer of a locally competitive algorithm, therefore, is to find what is the relevant information that must be communicated, and how to process it efficiently to obtain the desired output.

4.3 Discussion

The local competitiveness setting described above is specialized for the two clock synchronization problems given. It is straightforward to generalize it for other optimization problems along the following lines. The analog for local clock would be some source that generates inputs; local time at a point would be replaced by the cumulative input up to that point. The non-interfering filtering property remains unchanged, which means on one hand that a locally competitive algorithm works for any given view, and on the other hand that it does not generate messages on its own. The local competitiveness definition can be generalized using any positive valued *target function* that measures the quality of the output.

Approaches similar to local competitiveness were used in the past. For example, see the "best effort" algorithm of Fischer and Michael [9] for database management. (It may be interesting to note that the algorithm in [9] uses synchronized clocks.) Some other work was done by Ajtai *et al.* [2], after our preliminary paper was published [29]. Loosely speaking, in [2] they consider a shared memory system, where an execution is a sequence of processor accesses to the shared memory. The order by which processors take steps is given by an arbitrary *schedule*. A *task* is defined as a predicate over the output values, and a task is said to be *completed* when this predicate is satisfied. In the formulation of [2], the competitive factor of an algorithm is the maximum, over all schedules, of the total number of steps taken by the algorithm until the task is completed, divided by the minimal number of steps required by any correct algorithm to complete the task, under the same schedule. Our approach differs in a few technical aspects. First, our model is message passing and not shared memory; hence the analog of their "schedule" is our "view." Secondly, we consider an optimization problem, where output must be produced at all times. Hence the quantity of interest for us is a target function defined over the output values, whereas in [2], the output values are of no interest (provided they are correct), and the implicit target function is the number of steps required to produce the output.

Nevertheless, the local competitiveness approach is not widely accepted. One possible reason to reject it is that a locally competitive algorithm does not give an absolute guarantee but only a relative one. For example, in our formulation a locally competitive algorithm never initiates transmission of a message by itself. If no message is sent by the send module, then the optimal algorithm may be trivial since the best possible output is trivial. This example points to a deeper problem in system design (shared also by the classical competitiveness model of [32, 23]): the question is to determine what is the input for the algorithm, and what is under the control of the algorithm.

The reader should note, however, that a locally competitive algorithm must do well on all cases. In addition, the local competitiveness approach enables us to compare the performance of algorithms on equal grounds. For example, consider a system which is a ring of processors, and one algorithm that sends messages only clockwise, and another that sends messages only counterclockwise. It seems that the two algorithms are incomparable on a per-view basis, since effectively they run on different systems. However, if the algorithms are locally competitive, they must give good results on both cases.

Another possible objection to the concept of local competitiveness is the validity of the "non-interfering filtering" assumption. This assumption says, among other things, that the transmission time of a message is independent of the message added by the CSA, and that CSAs relay messages between the send module and the network links instantaneously. Strictly speaking, this assumption is false in any physical system. Nevertheless, we argue that the non-interfering filtering assumption can serve as a reasonable approximation of reality so long as the blowup in message size, and the computation resources required by the CSA are negligible.

We believe that the philosophy behind the concept of local competitiveness best suits network-maintenance protocols, e.g., topology update, or other routing protocols, where there is always something to be done. It is interesting to observe that in real networks, the message delivery system appends "headers" to messages to facilitate delivery. Ideal locally competitive algorithms would use such headers, extending them only slightly.

Summary

In this chapter we defined the synchronization tasks we consider in this thesis and the way we evaluate the performance of algorithms that solve them.

We defined the problem of *external synchronization*, in which all processors are trying to acquire tight bounds on the reading of one designated processor whose clock is drift-free. In the problem of *internal synchronization*, all processors need to make a distinguished action in the smallest possible interval of real time. For each problem we defined the system architecture, correctness requirement, and the measure of tightness.

The quality of a synchronization algorithm is measured by its *local competitiveness*. The local competitiveness of an algorithm is the maximal ratio between the tightness it produces at any point, and the best possible tightness for the given local view at that point. The concept of local competitiveness can be viewed as a combination of the per-execution evaluation approach of [3], competitive analysis [32, 23], and the causality partial order [16].

We argued that this approach can be of independent interest as a method for evaluating distributed optimization tasks. We compared the concept of local competitiveness with the approach of [2], and we discussed some of its advantages and disadvantages.

Chapter 5

The Basic Result

The starting point for this chapter is the following problem: given two points in an execution of a clock synchronization system, find the tightest bounds on the real time that elapses between their occurrence. The means by which this task is to be accomplished is the CSA modules. The "input" available to the CSA modules consists of the events that occurred in the system with their local time of occurrence (i.e., the *view* of the execution), and the standard *bounds mapping* that represents the system timing specification for that view. Hence the task can be solved if we can find the set of executions with the given view.

Our strategy to solve this problem is to reformulate the setting in graph-theoretic language, and solve a more general abstract problem. We first abstract views as labeled directed graphs, which we call *v-graphs*; the only attribute a point has in a v-graph is its local time. We also abstract patterns as labeled directed graphs, which we call *p-graphs*; in p-graphs, a point has both local and real time. Bounds mapping is now an abstract function that maps pairs of adjacent points in v-graphs to numbers. Using bounds mapping and v-graphs, we obtain weighted directed graphs we call *synchronization graphs*. Then, in Theorems 5.4 and 5.5, we prove a characterization of the set of p-graphs that have a given v-graph and satisfy a given bounds mapping, in terms of distances in the derived synchronization graph. These results are independent of the particular interpretation, but to aid intuition, our development is accompanied with an an example of an execution of a clock synchronization system.

Then, in the main results of this chapter, we specialize to the case of views and patterns of clock synchronization systems. In Theorems 5.6 and 5.7, we use Theorems 5.4 and 5.5

in conjunction with Theorem 3.4, and prove that the relation proven for p-graphs and synchronization graphs holds for patterns of executions of synchronization systems and the synchronization graphs derived from the views and bounds mapping. Using Theorem 3.2, we also derive a corollary for local views (Theorem 5.8).

Philosophically, synchronization graphs can be viewed as an extension of the graphs used by Lamport to describe executions of completely asynchronous systems [16]. Lamport's graphs are unweighted, and the main property of interest regarding a pair of points is whether one is reachable from the other. Reachability expresses the fact that in all possible executions which have that graph, one point occurs before the other. By contrast, we consider systems with clocks, and define graphs which are weighted. The main property of interest regarding two points is the distance between them: this distance expresses bounds on the real time that elapsed between their occurrence which is satisfied by all executions with that synchronization graph.

This chapter is organized as follows. In Section 5.1 we present the notions of v-graphs, p-graphs, synchronization graphs and prove a relation between these abstract concepts. In Section 5.2 we derive the results for clock synchronization systems.

5.1 Synchronization Graphs

In this section, we define the notions of v-graphs, p-graphs, and synchronization graphs. V-graphs and p-graphs are abstractions of views and patterns, respectively. We give a natural correspondence between the abstract graphs concepts and their counterparts in clock synchronization systems.

We define the key concept of *synchronization graphs*, which are weighted directed graphs, derived from v-graphs and bounds mappings for these graphs; synchronization graphs will be our main tool in analyzing executions of clock synchronization systems. The main results in this section relate p-graphs to the synchronization graph. The development in this section is self-contained; to help the reader in understanding the motivation for the concepts, we give a running example from our intended application domain, namely clock synchronization systems.

We start by defining the notion of v-graphs.

Definition 5.1 *A* v-graph *is a pair* $(G, local\_time)$*, where* $G = (V, E)$ *is a directed graph*

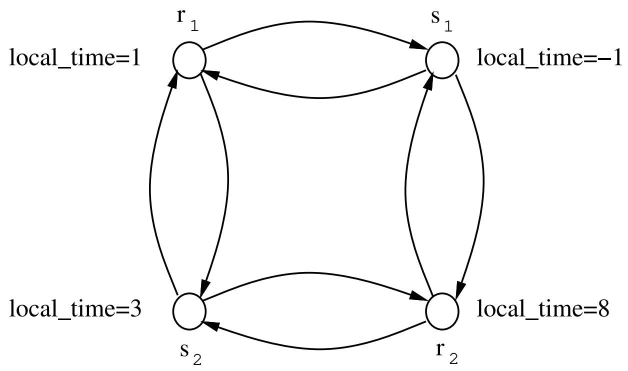

Figure 5-1: *An example of a v-graph.*

with $(p, q) \in E$ *if and only if* $(q, p) \in E$*, and local_time is a function that associates a finite real number with each point* $p \in V$*. For any two points* $p, q \in V$*, we define* $virt\_del(p, q) = local\_time(p) - local\_time(q)$*. A* **bounds mapping for a v-graph** *is a function that assigns a number* $B(p, q) \in \mathbf{R} \cup \{\infty\}$ *to each arc* $(p, q) \in E$*.*

The natural correspondence: views and v-graphs. Before we proceed to analyze view graphs, we describe the way v-graphs can be obtained from views of clock synchronization systems. Recall that a view $\mathcal{V}$, as defined in Def. 3.5, is a graph, where each point is labeled by an action name and local time of occurrence. Notice that by adding for each arc (p, q) in a view another arc (q, p), we obtain a v-graph. In the resulting v-graph there is some additional information attached to each point (i.e., the name of the associated action or null point), but this is irrelevant for our treatment of v-graphs. We call the above mapping from views to v-graphs the *natural correspondence*. In the sequel, points will be used to denote both points in view graphs and in views, where the interpretation is clear by the context.

The natural correspondence enables us to use bounds mappings for views as bounds mapping for v-graphs (recall that a bounds mapping for a view is a function that assigns an upper bound to the difference in real time between the occurrence of any two adjacent points in $\mathcal{V}$, see Def. 3.10). Under the natural correspondence, a bounds mapping for a view $\mathcal{V}$ applies also to pairs of adjacent points in the v-graph of $\mathcal{V}$.

Example. Consider a system with two processors u and v, and suppose that u has a drift-free clock, and v has a $(0.5, 1.5)$-clock. Consider the following scenario.

(1) u sends a message m_1 to v at local time -1, such that m_1 is guaranteed to be delivered within no less than 2 time units, and no more than 3 time units.

(2) m_1 is received at v at local time 1.

(3) v sends a message m_2 to u at local time 3, such that m_2 is guaranteed to be delivered within no less than 5 time units, and there is no upper bound on its transmission time.

(4) m_2 is received at u at local time 8.

The short description above provides sufficient detail to define a view, a v-graph, and a bounds mapping. Let s_1, s_2 denote the send points of m_1 and m_2, respectively, and let r_1, r_2 be their respective receive points. The corresponding v-graph is depicted in Figure 5-1. Also, we have that

$$
\begin{array}{rclcrcl}
virt\_del(s_1, r_1) & = & -2 & \qquad & virt\_del(r_1, s_1) & = & 2 \\
virt\_del(s_2, r_2) & = & -5 & & virt\_del(r_2, s_2) & = & 5 \\
virt\_del(s_1, r_2) & = & -9 & & virt\_del(r_2, s_1) & = & 9 \\
virt\_del(s_2, r_1) & = & 2 & & virt\_del(r_1, s_2) & = & -2
\end{array}
$$

Let B' denote the standard bounds mapping for the given view. Using Def. 3.11 we calculate the values of B'. We get

$$
\begin{array}{rclcrcl}
B'(s_1, r_1) & = & -2 & \qquad & B'(r_1, s_1) & = & 3 \\
B'(s_2, r_2) & = & -5 & & B'(r_2, s_2) & = & \infty \\
B'(s_1, r_2) & = & -9 & & B'(r_2, s_1) & = & 9 \\
B'(s_2, r_1) & = & 4 & & B'(r_1, s_2) & = & -4/3
\end{array}
$$

We shall return to this example as we develop the analysis. ∎

For the remainder of this section, we fix a v-graph $\beta = (G, local\_time)$ where $G = (V, E)$, and a bounds mapping B for β.

Our next step is to define the concept of a p-graph as an extension of a v-graph, analogous to the way a pattern is an extension of a view.

Definition 5.2 *A* p-graph *with view β is a triple $\alpha = (G, local\_time, now_\alpha)$, where $(G, local\_time) = \beta$, and now_α is a function that associates a non-negative finite real number with each* *point* $p \in V$.[1] *A p-graph* α *with view* β *is said to* satisfy B *if for all* $(p, q) \in E$ *we have* $act\_del_\alpha(p, q) \stackrel{\text{def}}{=} now_\alpha(p) - now_\alpha(q) \leq B(p, q)$.

For a given p-graph, we define the key concepts of *offsets*.

Definition 5.3 (Offset) *Let* p *be a point in a p-graph* $\alpha = (G, local\_time, now_\alpha)$. *The* absolute offset *of* p *is*

$$\delta_\alpha(p) = now_\alpha(p) - local\_time(p) \ .$$

For any other point q *in* α, *the* relative offset *of* p *from* q *is*

$$\delta_\alpha(p, q) = \delta_\alpha(p) - \delta_\alpha(q) \ .$$

We omit subscripts when no confusion arises.

The natural correspondence: patterns and p-graphs. The natural correspondence defined above for views applies also for patterns. This way, given a pattern $\mathcal{P}$ as defined in Def. 3.5, its p-graph α is naturally defined. Moreover, using the natural correspondence, the notions of absolute and relative offsets, defined over the points of α, are also defined over the points of $\mathcal{P}$, and we have that $\delta_\mathcal{P}(p) = \delta_\alpha(p)$ and $\delta_\mathcal{P}(p, q) = \delta_\alpha(p, q)$ for all points p, q. As an aside, notice that if we know local time of two points in an execution, then bounding the real time that elapses between their occurrences is equivalent to bounding their relative offset.

Before we proceed, we state two properties of relative offsets.

Lemma 5.1 *Let* p, q, r *be any three points of a given p-graph. Then*

1. $\delta(p, q) = -\delta(q, p)$ *(antisymmetry).*

2. $\delta(p, q) = \delta(p, r) + \delta(r, q)$ *(chain rule).*

Proof: Immediate from definitions. ∎

[1]The v-graph β and the bounds mapping B are fixed in this section; since we shall be dealing with many possible patterns, the *now* function is subscripted by the pattern's name.

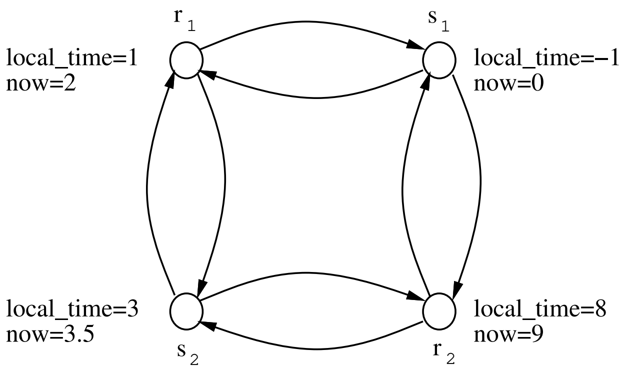

Figure 5-2: *An p-graph with view as in Figure 5-1.*

Example (continued). Figure 5-2 shows a p-graph whose view is given in Figure 5-1. It is easy to verify that this p-graph satisfies the bounds mapping B'. Let us compute the offsets for this p-graph. First, we compute the absolute offsets of the points. We get that

$$\begin{aligned} \delta(s_1) &= 1 & \qquad \delta(r_1) &= 1 \\ \delta(s_2) &= 0.5 & \qquad \delta(r_2) &= 1 \end{aligned}$$

Now we compute the relative offsets of pairs of points (reversing the order of the points negates the sign):

$$\begin{aligned} \delta(s_1, r_1) &= 0 & \qquad \delta(s_1, r_2) &= 0 \\ \delta(s_1, s_2) &= 0.5 & \qquad \delta(r_1, s_2) &= 0.5 \\ \delta(r_1, r_2) &= 0 & \qquad \delta(s_2, r_2) &= -0.5 \end{aligned}$$

∎

Next, based on the v-graph $\beta = (G, \mathit{local\_time})$ and the bounds mapping B, we introduce weights for the arcs of G. The resulting weighted graph, called the *synchronization graph*, is our primary tool for analyzing executions of clock synchronization systems.

Definition 5.4 (Synchronization Graph) *The* synchronization graph *generated by the v-graph β and its bounds mapping B is a weighted directed graph $\Gamma = (V, E, w)$, where $(V, E) = G$, and $w(p, q) = B(p, q) - \mathit{virt\_del}(p, q)$ for all $(p, q) \in E$.*

Example (continued). The synchronization graph generated by the v-graph in Figure 5-1 and B' is depicted in Figure 5-3. ∎

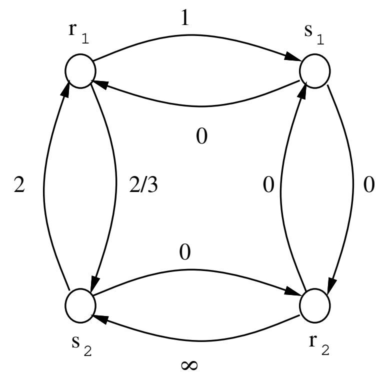

Figure 5-3: *The synchronization graph generated by the v-graph in Figure 5-1 and B'.*

We now arrive at the main theme of this section, which is to study the connection between p-graphs and the synchronization graph. The following lemma states the basic property of arc weights of the synchronization graph. (Notice that since we have fixed β and B, we also have now Γ fixed for the remainder of the section.)

Lemma 5.2 *If a given p-graph with v-graph β satisfies B, then $\delta(p,q) \in [-w(q,p)\,,\, w(p,q)]$.*

Proof: Since the p-graph satisfies B, we have that $act\_del(p,q) \le B(p,q)$ and $act\_del(q,p) \le B(q,p)$, and hence $act\_del(p,q) \in [-B(q,p), B(p,q)]$. Therefore,

$$\begin{array}{rcll}
\delta(p,q) & = & (now(p) - local_time(p)) - (now(q) - local_time(q)) & \textit{by definition} \\
& = & act_del(p,q) - virt_del(p,q) & \textit{rearranging} \\
& \in & [-B(q,p) - virt_del(p,q)\ ,\ B(p,q) - virt_del(p,q)] & \textit{by assumption} \\
& = & [-B(q,p) + virt_del(q,p)\ ,\ B(p,q) - virt_del(p,q)] & \textit{by antisymmetry} \\
& = & [-w(q,p)\,,\, w(p,q)]\ . & \textit{by definition}
\end{array}$$

∎

Our next step is to look at the natural concept of distance between points in the synchronization graph. Formally, we have the following (standard) definition.

Definition 5.5 *The* weight of a path *$\theta = p_0, p_1 \ldots, p_k$ in a weighted graph $\Gamma = (V, E, w)$ is $w(\theta) = \sum_{i=1}^{k} w(p_{i-1}, p_i)$. A path from p to q is a* shortest path *if its weight is minimum among all paths from p to q. The* distance *from p to q, denoted $d(p,q)$, is the weight of a shortest path from p to q, or ∞ if there is no path from p to q.*

Notice that the distances are not well defined if Γ has cycles with negative weights. The next lemma gives a sufficient condition for Γ to have no negative-weight cycles.

Lemma 5.3 *If there exists a p-graph α with v-graph β such that α satisfies B, then Γ has no negative weight cycles.*

Proof: Let $\theta = \langle p_0, p_1, \ldots, p_{k-1}, p_k = p_0 \rangle$ be any directed cycle in Γ. Then

$$\begin{array}{rcll} w(\theta) & = & \sum_{i=1}^{k} w(p_{i-1}, p_i) & \\ & \geq & \sum_{i=1}^{k} \delta_\alpha(p_{i-1}, p_i) & \textit{by Lemma 5.2} \\ & = & \delta_\alpha(p_0, p_0) & \textit{by Lemma 5.1} \\ & = & 0 \ . & \textit{by definition} \end{array}$$

∎

We now arrive at the first result for the problem of determining the set of p-graphs that satisfy B and have v-graph β. The following theorem characterizes these p-graphs in terms of all distances in the synchronization graph.

Theorem 5.4 *A p-graph α with v-graph β satisfies B if and only if for any two points $p, q \in V$ in the synchronization graph, $\delta_\alpha(p,q) \leq d(p,q)$.*

Proof: Let α be a p-graph with v-graph β. Assume first that α satisfies B, i.e., for any $(p,q) \in E$ we have $act\_del_\alpha(p,q) \leq B(p,q)$. We show that $\delta_\alpha(p,q) \leq d(p,q)$ for any $p, q \in V$. In case that there is no path connecting p and q, we have $d(p,q) = \infty$ and we are done trivially. Otherwise, consider any shortest path $p = p_0, \ldots, p_k = q$ from p to q. Then we have that

$$\begin{array}{rcll} \delta_\alpha(p,q) & = & \sum_{i=0}^{k-1} \delta_\alpha(p_i, p_{i+1}) & \textit{by Lemma 5.1} \\ & \leq & \sum_{i=0}^{k-1} w(p_i, p_{i+1}) & \textit{by Lemma 5.2} \\ & = & d(p,q) & \textit{by definition} \end{array}$$

proving the "only if" part of the theorem.

Conversely, assume that for any two points $p, q \in V$, we have that $\delta_\alpha(p,q) \leq d(p,q)$. We prove that α satisfies B. Let $(p,q) \in E$. By definitions of arc weights and distances, we have that $B(p,q) - virt\_del(p,q) = w(p,q) \geq d(p,q)$. Hence, by assumption, we get $B(p,q) -$

$virt\_del(p,q) \geq d(p,q) \geq \delta_\alpha(p,q) = act\_del_\alpha(p,q) - virt\_del(p,q)$. Adding $virt\_del(p,q)$ to both sides, we get $B(p,q) \geq act\_del_\alpha(p,q)$, as desired. ∎

Example (continued). The distances in the synchronization graph of Figure 5-3 are given by

$$\begin{array}{rclcrcl}
d(s_1,r_1) & = & 0 & \qquad & d(r_1,s_1) & = & 2/3 \\
d(s_1,s_2) & = & 2/3 & & d(s_2,s_1) & = & 0 \\
d(s_1,r_2) & = & 0 & & d(r_2,s_1) & = & 0 \\
d(s_2,r_1) & = & 0 & & d(r_1,s_2) & = & 2/3 \\
d(s_2,r_2) & = & 0 & & d(r_2,s_2) & = & 2/3 \\
d(r_2,r_1) & = & 0 & & d(r_1,r_2) & = & 2/3
\end{array}$$

As the reader may verify, for the pattern of Figure 5-2 we have that $\delta(p,q) \in [-d(q,p), d(p,q)]$ for all points p, q in the view. ∎

Before we state the next theorem (which is the major result of this section), we define the following technical terms. The complicated-looking definition is due to the fact distances may be infinite.

Definition 5.6 *Suppose Γ has no negative weight cycles. Let α be a p-graph with v-graph β, let $p_0 \in V$, and let $N > 0$.*

*(1) α is an N-**p-graph from** p_0 if for all $q \in V$: if $d(p_0,q) < \infty$ then $\delta_\alpha(p_0,q) = d(p_0,q)$, and otherwise $\delta_\alpha(p_0,q) > N$.*

*(2) α is an N-**p-graph to** p_0 if for all $q \in V$: if $d(q,p_0) < \infty$ then $\delta_\alpha(q,p_0) = d(q,p_0)$, and otherwise $\delta_\alpha(q,p_0) > N$.*

The offsets in an N-p-graph from p_0 are the distances from p_0, with infinite distances replaced by offsets larger than N, and analogously for an N-p-graph to p_0. Using these notions, we state the following theorem.

Theorem 5.5 *Suppose Γ has no negative-weight cycles. Then for any point $p_0 \in V$, and for any finite number $N > 0$, there exist p-graphs α_0 and α_1, such that both have view β, both satisfy B, and such that α_0 is an N-p-graph to p_0, and α_1 is an N-p-graph from p_0.*

Proof: To prove the theorem, we first construct a related graph Γ^* in which all distances are finite. Based on Γ^*, we define p-graphs α_0 and α_1, and then show that α_0 and α_1 have the required properties.

To construct Γ^*, we first choose a number M that is sufficiently large so as to satisfy

$$M \;>\; N + \sum_{\substack{(p,q)\in E \\ 0<w(p,q)<\infty}} w(p,q) \;-\; \sum_{\substack{(p,q)\in E \\ -\infty<w(p,q)<0}} w(p,q)\;.$$

Using M, we augment Γ with extra arcs as follows. For each pair of points p, q such that $d(p,q) = \infty$, we add an *artificial arc* (p,q) with weight M. Call the resulting augmented graph Γ^*, and denote its distance function by d^*. The following claim shows the connection between the distances in Γ^*, the distances in Γ, and N.

Claim A. *For all $p, q \in V$, if $d(p,q) < \infty$, then $d^*(p,q) = d(p,q)$, and if $d(p,q) = \infty$, then $N < d^*(p,q) < \infty$.*

Proof of Claim A: We start (for future reference) with an inequality that follows directly from the choice of M. Let X and Y denote arbitrary subsets of the arcs of Γ with finite weights. Then

$$M + \sum_{(p,q)\in X} w(p,q) > \max\left\{ N\,, \sum_{(p,q)\in Y} w(p,q) \right\} \tag{5.1}$$

Next, we argue that the augmented graph Γ^* has no negative weight cycles. Suppose, for contradiction, that there exists some negative weight cycle in Γ^*. Then one of arcs of the cycle, say (p,q), must be an artificial arc, and there must be a simple directed path Z in Γ^* from q to p with total weight w_Z such that $M + w_Z < 0$. Let $\underline{w}_Z$ be the sum of *negative* weight arcs of Z. Clearly, $\underline{w}_Z \le w_Z$. Also, by Eq. (5.1), we have that the sum of M and the weights of any subset of arcs of Γ is at least N. Since all artificial arcs have positive weight, we know that $\underline{w}_Z$ is the sum of weights of arcs from Γ. Therefore we have that $M + w_Z \ge M + \underline{w}_Z > N > 0$, a contradiction.

To show that the finite distances in Γ remain invariant in Γ^*, we first note that since Γ is a subgraph of Γ^*, it must be the case for all $p, q \in V$ that $d^*(p,q) \le d(p,q)$. Suppose for contradiction that for some $p, q \in V$ with $d(p,q) < \infty$ we have $d^*(p,q) < d(p,q)$. Since, as we showed above, Γ^* has no negative-weight cycles, we may assume that there exists a simple path in Γ^* with length $d^*(p,q)$. Clearly, one of its arcs is artificial. However, by Eq. (5.1), this means that the total weight of that path is larger than the total weight of any finite-weight simple path in Γ, a contradiction.

Finally, let $p, q \in V$ be such that $d(p,q) = \infty$. Clearly $d^*(p,q) < \infty$ by virtue of the artificial arc (p, q). To see that $d^*(p, q) > N$, consider any simple path from p to q. As before, this path contains at least one artificial arc, and therefore its total weight is at least M plus all negative weights of Γ. Using Eq. (5.1), we get that the total weight of the path is greater than N. ▮

We now define the p-graphs α_0 and α_1 explicitly. Since their view is given, the events and their local times are already fixed; we complete the construction by specifying the *now* mappings of the p-graphs. Let L be a number such that

$$L > \min_{q \in V} \{local\_time(q) + d^*(q, p_0), local\_time(q) - d^*(p_0, q)\} \ .$$

For all $q \in V$, we set

$$\begin{aligned} now_{\alpha_0}(q) &= L + local\_time(q) + d^*(q, p_0) \\ now_{\alpha_1}(q) &= L + local\_time(q) - d^*(p_0, q) \end{aligned}$$

(The additional term L guarantees that all *now* values are positive.) By the construction, for all $q \in V$ we have

$$\begin{aligned} \delta_{\alpha_0}(q) &= now_{\alpha_0}(q) - local\_time(q) \\ &= L + (d^*(q, p_0) + local\_time(q)) - local\_time(q) \\ &= L + d^*(q, p_0) \ . \end{aligned} \tag{5.2}$$

Since $d^*(p_0, p_0) = 0$, we have that $\delta_{\alpha_0}(p_0) = L$, and therefore $\delta_{\alpha_0}(q, p_0) = \delta_{\alpha_0}(q) - \delta_{\alpha_0}(p_0) = d^*(q, p_0)$. Similarly, we obtain that $\delta_{\alpha_1}(p_0, q) = -d^*(p_0, q)$. Therefore, by Claim A, α_0 is an N-p-graph to p_0 and α_1 is an N-p-graph from p_0. The following claim completes the proof of the theorem.

Claim B. *The p-graphs α_0 and α_1 defined above satisfy the bounds mapping B.*

Proof of Claim B: By Theorem 5.4, it is sufficient to prove that for all $p, q \in V$, $\delta_{\alpha_0}(p, q) \leq d(p, q)$. So let p and q be arbitrary points in the synchronization graph. In what follows, we consider Γ^*, the graph defined above. Since $d^*(p, q) \leq d(p, q)$, it is sufficient to prove that $\delta_{\alpha_0}(p, q) \leq d^*(p, q)$.

Let R be any shortest path from p to q. Consider the path obtained by following the

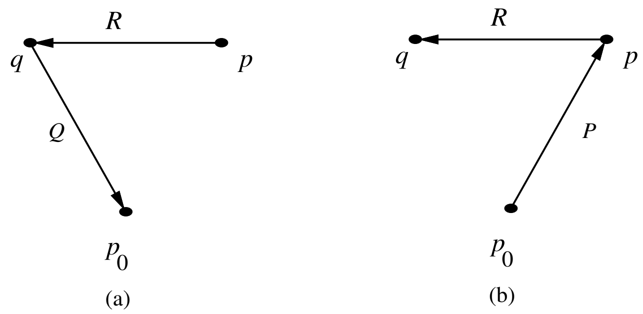

Figure 5-4: *Scenarios considered in the proof of Claim B. R is a shortest path from p to q, P is a shortest path from p_0 to p, and Q is a shortest path from q to p_0.*

arcs of R from p to q, and then the arcs of a shortest path from q to p_0 (see Figure 5-4(a)). This path leads from p to p_0, and hence $d^*(p,q) + d^*(q,p_0) \geq d^*(p,p_0)$. It follows from Eq. (5.2) and the definition of relative offsets that

$$\begin{aligned} d^*(p,q) &\geq d^*(p,p_0) - d^*(q,p_0) \\ &= \delta_{\alpha_0}(p) - \delta_{\alpha_0}(q) \\ &= \delta_{\alpha_0}(p,q)\ . \end{aligned}$$

I.e., for all $p,q \in V$, $\delta_{\alpha_0}(p,q) \leq d^*(p,q)$, and therefore, by Theorem 5.4, we conclude that α_0 satisfies the given bounds mapping B, as desired.

The proof for α_1 is analogous. We consider a shortest path R connecting two arbitrary points p and q. To show that its weight $d^*(p,q)$ is at least $\delta(p,q)$, we look at the path depicted in Figure 5-4(b), consisting of a shortest path P from p_0 to p, followed by R. As before, we have that $d^*(p_0,p) + d(p,q) \geq d^*(p_0,q)$, and hence we get

$$\begin{aligned} d^*(p,q) &\geq d^*(p_0,q) - d^*(p_0,p) \\ &= -\delta_{\alpha_1}(q) + \delta_{\alpha_1}(p) \\ &= \delta_{\alpha_1}(p,q)\ . \end{aligned}$$

Therefore, $\delta_{\alpha_1}(p,q) \leq d^*(p,q)$ for all points $p,q \in V$, and applying Theorem 5.4 shows that α_1 satisfies B, as desired. ∎

This completes the proof of Theorem 5.5. ∎

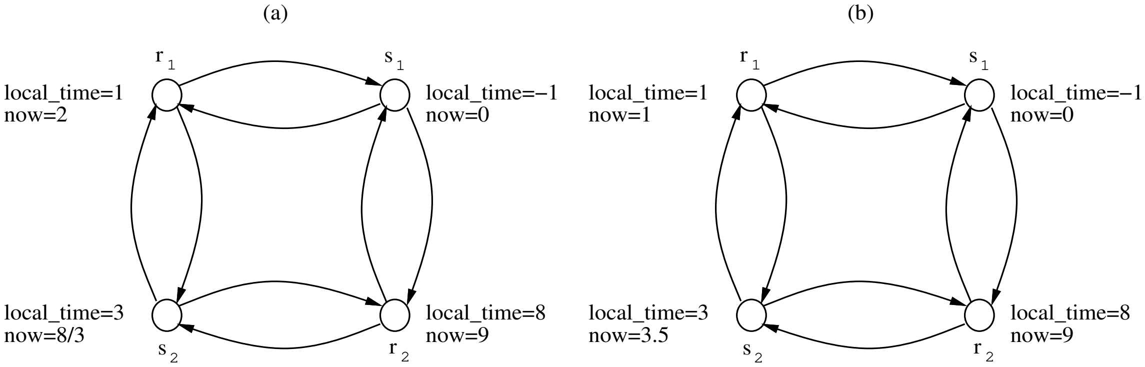

Figure 5-5: *Assuming $now(r_2) = 9$, (a) is a pattern from r_2, and (b) is a pattern to r_2.*

Example (conclusion). Using the distances calculated above for the synchronization graph of Figure 5-3, we can compute patterns from and to the point r_2. Since the definition of these patterns only specifies relative offsets, we fix $now(r_2) = 9$ (agreeing with the pattern of Figure 5-2 at this point). The resulting pattern from r_2 is given in Figure 5-5 (a), and the resulting pattern to r_2 is given in Figure 5-5 (b). It is a simple matter to verify that both patterns have the view depicted in Figure 5-1, and they satisfy the bounds mapping B'. One conclusion from these patterns is that an observer located at r_2, with access only to the view and the bounds mapping, cannot determine the time of occurrence of s_1 with tightness greater than $7/2 - 8/3 = 5/6$ real time units, since both patterns depicted in (a) and (b) describe a possible scenario. ∎

5.2 Interpretation in Clock Synchronization Systems

Theorems 5.4 and 5.5 describe a relation between p-graphs and synchronization graphs. In this section we apply these results to executions of clock synchronization systems. In other words, in this section we deal with views and patterns of executions of clock synchronization systems (as defined in Section 3.2.1), instead of abstract v-graphs and p-graphs, respectively. We apply, in a straightforward fashion, the theorems of Section 5.1, in conjunction with Theorem 3.4, using the natural correspondence (defined in Section 5.1), which maps views and patterns to v-graphs and p-graphs, respectively. Before we state and prove the (somewhat technical, albeit straightforward) theorems, we make two comments about the results.

1. By our definitions of clock synchronization systems, synchronization graphs can

be used under a wide variety of assumptions. In particular, they can be used to model executions where messages may be lost, delivered out of order, or duplicated by the communication links; they can be used to model broadcast channels; they can be used for the case of processor and link crashes; and by our definition of bounds mapping, they can also be used to model clock drift bounds that may change over time.

2. The essential assumptions in our analysis are the following. First, if an offset can be a value a and a value b, then it can also be any value in between. This rules out scenarios in which the offset might be *either* a or b (as might be the case for messages over framed communication links, or clocks with fixed but unknown rate). Removing this assumption will result in a constraint system which is not even a linear program, and cannot be represented as distance computation techniques. The second important assumption in our analysis is that "patterns satisfy the bounds mapping," that is to say, the system behaves according to its specification. As indicated by Lemma 5.3 (and explained in Chapter 9), synchronization graphs are still useful in some limited sense in the case that executions do not satisfy the bounds mapping.

We now proceed with applying the analysis of Section 5.1 to clock synchronization systems. We recall that under the natural correspondence, each arc (p, q) in a view is replaced by a pair of arcs (p, q) and (q, p) in the corresponding v-graph, and that local time attributes, bounds mapping values (and real times in p-graphs) remain unchanged. Under the natural correspondence, the notion of offsets that was defined for p-graphs (Def. 5.3) applies to executions and patterns of clock synchronization systems. The offset between two points p, q in a pattern $\mathcal{P}$ is

$$\begin{aligned}
\delta_{\mathcal{P}}(p,q) &= \delta_{\mathcal{P}}(p) - \delta_{\mathcal{P}}(q) \\
&= (now_{\mathcal{P}}(p) - local_time_{\mathcal{P}}(p)) - (now_{\mathcal{P}}(q) - local_time_{\mathcal{P}}(q)) \\
&= act_del_{\mathcal{P}}(p,q) - virt_del_{\mathcal{P}}(p,q)
\end{aligned}$$

It follows that if we know the local times of occurrence of p and q, then bounding the real time that elapses between their occurrences is equivalent to bounding $\delta(p, q)$. This seems to capture a useful quantity in any synchronization problem. The theorems in this section provide us with a characterization of the bounds on the offset in a pattern with a given view and bounds mapping, and hence they are useful in analyzing synchronization problems.

First, we state the theorem that is the key in proving *correctness* of clock synchronization algorithms.

Theorem 5.6 *Let $\mathcal{V}$ be a view of an execution of a clock synchronization system $\mathcal{S}$, and let B be the standard bounds mapping for $\mathcal{V}$. Let Γ be the synchronization graph generated by the v-graph of $\mathcal{V}$ and B, and let d_Γ be its distance function. Let $\mathcal{P}$ be any pattern with view $\mathcal{V}$. Then there exists an execution e' of $\mathcal{S}$ whose pattern is $\mathcal{P}$ if and only if for any two points p, q in $\mathcal{P}$, we have $\delta_{\mathcal{P}}(p, q) \leq d_\Gamma(p, q)$.*

Proof: Suppose first that there exists an execution e' of $\mathcal{S}$ with pattern $\mathcal{P}$, and consider its p-graph α. Since by assumption e' is an execution of $\mathcal{S}$, $\mathcal{P}$ satisfies B, and hence α satisfies B. Therefore, by Theorem 5.4, for any two points p, q in α, $\delta_\alpha(p, q) \leq d_\Gamma(p, q)$, and since $\delta_\alpha(p, q) = \delta_{\mathcal{P}}(p, q)$, we are done in this case.

Suppose now that for a pattern $\mathcal{P}$ with view $\mathcal{V}$, we have $\delta_{\mathcal{P}}(p, q) \leq d_\Gamma(p, q)$ for every pair of points p, q in $\mathcal{P}$. It follows that in the p-graph α of $\mathcal{P}$, $\delta_\alpha(p, q) \leq d_\Gamma(p, q)$ for every pair of points p, q. Hence, by Theorem 5.4, α satisfies B, and therefore $\mathcal{P}$ satisfies B. Finally, since $\mathcal{P}$ satisfies the standard bounds B, we may apply Theorem 3.4, and conclude that there exists an execution e' of $\mathcal{S}$ whose pattern is $\mathcal{P}$. ∎

Next, we present the theorem we shall use for proving lower bounds on the tightness achievable by synchronization algorithms. We first define the notions of N-patterns to and from a point. The definition is the equivalent of Def. 5.6 under the natural correspondence.

Definition 5.7 *Let Γ be a synchronization graph for a view $\mathcal{V}$, and let $\mathcal{P}$ be a pattern with view $\mathcal{V}$. Let α be the p-graph for $\mathcal{P}$ under the natural correspondence, and let p_0 be a point in α. For any $N > 0$, $\mathcal{P}$ is an N-*pattern from p_0 *if α is an N-p-graph from p_0, and it is an* N-pattern to p_0 *if α is an N-p-graph to p_0.*

The following theorem is the application of Theorem 5.5 to clock synchronization systems. Intuitively, it says that there exist indistinguishable executions of clock synchronization systems, where the offsets between a a given point and any other point are exactly the distances in the synchronization graph, and hence any synchronization algorithm must take these extreme cases into account.

Theorem 5.7 *Let $\mathcal{V}$ be a view of an execution e of a clock synchronization system $\mathcal{S}$ (possibly including null points), and let B be the standard bounds mapping for $\mathcal{V}$. Let Γ be the* *synchronization graph generated by the v-graph of $\mathcal{V}$ and by B, and let d_Γ be its distance function. Let p_0 be any point in $\mathcal{V}$. Then for any finite number $N > 0$, there exist executions e_0 and e_1 of $\mathcal{S}$, such that both have view $\mathcal{V}$, and such that the pattern of e_0 is an N-pattern to p_0, and the pattern of e_1 is an N-pattern from p_0. Moreover, for each CSA module C_v, the executions of C_v in e_0 and in e_1 are equivalent.*

Proof: First, note that since Γ is obtained from an execution of $\mathcal{S}$, its pattern $\mathcal{P}$ satisfies the standard bounds mapping B. From Theorem 5.6 we get that for any two points p, q in $\mathcal{P}$, $\delta_{\mathcal{P}}(p,q) \leq d_\Gamma(p,q)$; in particular, since $\delta_{\mathcal{P}}(p,p) = 0$ for all points p, we conclude that there are no negative-weight cycles in Γ. Hence we can apply Theorem 5.5, and get p-graphs α_0 and α_1 which are N-patterns to and from p_0, respectively, such that both satisfy B. Using the natural correspondence between $\mathcal{V}$ and its v-graph, we obtain from α_0 and α_1 patterns $\mathcal{P}_0$ and $\mathcal{P}_1$. Since α_0 and α_1 satisfy B, $\mathcal{P}_0$ and $\mathcal{P}_1$ satisfy B too. We can therefore apply Theorem 3.4, and the result follows. ∎

We also state a variant of Theorem 5.7 used for locality-oriented bounds.

Theorem 5.8 *Let $\mathcal{V}$ be a view of an execution e of a clock synchronization system $\mathcal{S}$ (possibly including null points), and let p_0 be any point in $\mathcal{V}$. Let B be the standard bounds mapping for the local view $\texttt{prune}(\mathcal{V}, p_0)$, and let Γ be the synchronization graph generated by $\texttt{prune}(\mathcal{V}, p_0)$ and B, and let d_Γ be its distance function. Then for any finite number $N > 0$, there exist executions e_0 and e_1 of $\mathcal{S}$, such that both have view $\texttt{prune}(\mathcal{V}, p_0)$, and such that the pattern of e_0 is an N-pattern to p_0, and the pattern of e_1 is an N-pattern from p_0. Moreover, for each CSA module C_v, the executions of C_v in e_0 and in e_1 are equivalent.*

Proof: By Theorem 3.2, there exists an execution e' whose view is $\texttt{prune}(\mathcal{V}, p_0)$ and such that for each CSA module C_v, $\texttt{prune}(e|_{C_v}, p) = \texttt{prune}(e'|_{C_v}, p)$. The theorem therefore follows by applying Theorem 5.7 to e'. ∎

Summary

In this chapter we abstracted the notions of views and patterns using the notions of v-graphs and p-graphs. We defined the concept of *offsets* of points in patterns, which captures an elementary synchronization problem. Using the bounds mapping, we define the basic tool of our analysis, namely the *synchronization graphs.* Using the offsets, we proved a simple characterization of the patterns which have a given view and bounds mapping, in terms of distances in the synchronization graph derived from the view and the bounds mapping. In particular, our main results in this chapter show that the bounds on synchronization obtained by the distances in the synchronization graphs are the best bounds possible, in the sense that there exist patterns that have the given view, satisfy the given bounds mapping, and meet the distance bounds.

The concept of synchronization graphs, specialized appropriately, serves as the basis for analyzing specific synchronization problems in Chapters 6, 7 and 8. A few simple variants of synchronization graphs are described in Chapter 9.

Chapter 6

External Synchronization

In this chapter we study a particular variant of the synchronization problem, called external synchronization. Informally, in the external synchronization problem there is a distinguished processor called the *source processor*, which is equipped with a drift-free clock. The task of all other processors is to produce, at all states, an estimate (i.e., an interval) that contains the current reading of the source clock. The name is motivated by an implicit assumption that the source clock serves as a source of real time in the system. The length of the estimate interval is called the tightness of synchronization at that point.

In this chapter, we obtain a few results for the external synchronization task, using Theorems 5.6 and 5.8. First, we characterize the achievable tightness of external synchronization for any processor at any given time, in terms of distances in the appropriate synchronization graph. The general algorithm we present, which achieves optimal tightness always, is a full information protocol, and hence inefficient. By contrast, for the special case of drift-free clocks, we present an optimal algorithm which is extremely efficient (and simple). The latter algorithm compares favorably to the so-called round-trip technique, used by many practical algorithms. In the last section of this chapter, we present the main ideas in the round-trip technique, based on NTP (Network Time Protocol, the external synchronization protocol used over the Internet [26]).[1] We also explain why our technique is superior to the one used in NTP.

This chapter is organized as follows. In Section 6.1 we recall the definition of external synchronization, and make a few preliminary observations. In Section 6.2 we give lower

[1] We use a simplified version introduced in Section 3.1.5 under the name SNTP.

and upper bounds on the tightness of external synchronization in a general system, where the non-source clocks have arbitrary drift bounds and arbitrary message latency bounds. In Section 6.3 we give an efficient optimal algorithm for systems with drift-free clocks. We conclude in Section 6.4 with a description of the round-trip technique, and compare it with our algorithm.

6.1 Problem Statement and Preliminary Observations

We recall the definition of the external clock synchronization problem. There exists in the system a distinguished processor s, called the *source processor*, whose local clock is drift-free. Each CSA module has two *output variables*, denoted ext_L_v and ext_U_v. For any given state x in an execution of an external synchronization system, let $source\_time(x)$ denote the local time at the source in x. The correctness requirement for a processor v is that in every reachable state x, the output variables satisfy $source\_time(x) \in [ext\_L_v, ext\_U_v]$. The *tightness* of synchronization at processor v in some state is the difference between the output variables in that state:

$$\Theta_v = ext\_U_v - ext\_L_v \ .$$

As a preliminary step in our analysis, we state a general property of drift-free clocks.

Lemma 6.1 *Suppose that processor v has a drift-free clock, and let $\Gamma = (V, E, w)$ be a synchronization graph obtained from a view of some execution of the system and the standard bounds mapping. Then the distance in Γ between any two points that occur at v is 0.*

Proof: We first claim that for any two adjacent points q, q' that occur in v, we have $w(q, q') = 0$. This follows immediately from definitions: by Def. 2.5, $\underline{\varrho}_v = \overline{\varrho}_v = 1$; by Def. 3.11, we have $B(q, q') = virt\_del(q, q')/\overline{\varrho}_v = virt\_del(q, q')$; and hence, by Def. 5.4, we have $w(q, q') = B(q, q') - virt\_del(q, q') = 0$.

This claim implies that there exists a 0-weight path between any two points occurring at v, and hence, for any two points q_1, q_2 that occur at v, we have that $d(q_1, q_2) \leq 0$. Suppose now, for the sake of contradiction, that there exists a path P from q_1 to q_2 with negative weight. Since there exists a a path Q from q_2 to q_1 of weight 0, we conclude that the cycle obtained by "gluing" P and Q together has negative weight, contradicting Lemma 5.3. ∎

The meaning of Lemma 6.1 is as follows. Suppose that a processor v has a drift-free clock, and let p_0 be any point in the synchronization graph. Then the distance to p_0 from any point q that occurs at v, and the distance from p_0 to any point q that occurs at v is independent of the particular choice of q, so long as q occurs at v. In other words, all points that occur at a processor whose local clock is drift-free are equivalent for the distance function in the synchronization graph. It is convenient to refer in this case to a *superpoint* associated with a drift-free processor v, defined formally to be an arbitrary representative of the points that occur at v. From the perspective of patterns, we notice that for a processor v whose clock is drift-free, the absolute offsets of all the points that occur at v are the same, and hence the notion of relative offset between any point and the superpoint of v is well defined.

The source clock, by definition, is drift-free. Given a synchronization graph of an external synchronization system, we call the superpoint associated with the source the *source point*, an denote it by sp throughout this chapter.

6.2 Bounds on the Tightness of External Synchronization

In this section we prove matching upper and lower bounds on the tightness of algorithms for external synchronization. The lower bound is derived from Theorem 5.8, and the upper bound follows from Theorem 5.6.

We start by fixing the scenario and the notation. Throughout this section we are dealing with an execution of an external synchronization system; let v be a processor in the system, and let x be a state in the execution. We denote $T_{x,v} = \mathit{local\_time}_v(x)$, and denote by $p_{x,v}$ the point that occurs at v at local time $T_{x,v}$. (If there is more than one such point, we take the last one; if there is no such point, $p_{x,v}$ is a null point we introduce.) Further, we denote $\mathcal{V}_{x,v} = \texttt{prune}(\mathcal{V}, p_{x,v})$, i.e., $\mathcal{V}_{x,v}$ is the local view of the execution at v at local time $T_{x,v}$. Let $B_{x,v}$ denote the standard bounds mapping for $\mathcal{V}_{x,v}$. We use the synchronization graph $\Gamma_{x,v} = (V, E, w)$ generated by the view graph of $\mathcal{V}_{x,v}$ and $B_{x,v}$, and denote the distance function of $\Gamma_{x,v}$ by $d_{x,v}$. Finally, recall that sp denotes the source point of $\Gamma_{x,v}$.

We start with a simple lemma that bounds the local time at the source in state x, in terms of the local time at v, and the distances between $p_{x,v}$ and the source point in the corresponding synchronization graph.

Lemma 6.2 *For all states x and processors v,*

$$source\_time(x) \in [T_{x,v} - d_{x,v}(sp, p_{x,v}), T_{x,v} + d_{x,v}(p_{x,v}, sp)] \; .$$

Proof: Consider the synchronization graph Γ obtained from the full view and the standard bounds mapping of the execution, and let d be the distance function in Γ. Since $\Gamma_{x,v}$ is a subgraph of Γ, we have that for every pair of points p, q in $\Gamma_{x,v}$

$$d_{x,v}(p, q) \geq d(p, q) \tag{6.1}$$

Now, let δ be the offset function of the execution, and let $T_{x,s} = source\_time(x)$. Then we have that

$$\begin{aligned} T_{x,s} &= (T_{x,s} - now(x)) - (T_{x,v} - now(x)) + T_{x,v} \\ &= \delta(sp, p_{x,v}) + T_{x,v} && \textit{by definition of } \delta \\ &\in [T_{x,v} - d(p_{x,v}, sp), T_{x,v} + d(sp, p_{x,v})] && \textit{by Theorem 5.6} \\ &\subseteq [T_{x,v} - d_{x,v}(p_{x,v}, sp), T_{x,v} + d_{x,v}(sp, p_{x,v})] && \textit{by Eq. (6.1)} \end{aligned}$$

∎

We now state the lower bound on the tightness of external synchronization.

Theorem 6.3 *Let x be any state in an execution of an external clock synchronization system, and let v be any non-source processor. Then in x,*

$$[ext\_L_v, ext\_U_v] \; \supseteq \; [T_{x,v} - d_{x,v}(sp, p_{x,v}) \; , \; T_{x,v} + d_{x,v}(p_{x,v}, sp)] \; .$$

Proof: Consider first the case where x occurs before the first action in v. Then clearly in x we have $[ext\_L_v, ext\_U_v] = [-\infty, \infty]$, and since $\Gamma_{x,v}$ does not contain the source point, we also have $d_{x,v}(sp, p_{x,v}) = d_{x,v}(p_{x,v}, sp) = \infty$, and we are done. Assume for the rest of the proof that x occurs after the first action of v.

Suppose that $d_{x,v}(sp, p_{x,v}) < \infty$ and $d_{x,v}(p_{x,v}, sp) < \infty$. By Theorem 5.8 (applied with p_0 substituted by $p_{x,v}$), there exist executions e_0 and e_1 such that both have view $\mathcal{V}_{x,v}$, and such that for e_0 we have $\delta_0(p_{x,v}, sp) = -d_{x,v}(sp, p_{x,v})$ and for e_1 we have $\delta_1(p_{x,v}, sp) = d_{x,v}(p_{x,v}, sp)$. Let ST_0 and ST_1 denote the source time when the local time at v is $T_{x,v}$ in e_0 and e_1, respectively. By definition, we have that $ST_0 = T_{x,v} + \delta_0(p_{x,v}, sp) = T_{x,v} -$

$d_{x,v}(sp, p_{x,v})$, and similarly, $ST_1 = T_{x,v} + d_{x,v}(p_{x,v}, sp)$. Moreover, Theorem 5.8 says that the basic state of the CSA module at v at local time $T_{x,v}$ is the same in the original execution, in e_0 and in e_1. Since the output variables of a CSA are part of its basic state component, it follows from the correctness requirement for external synchronization that in x,

$$[ext\_L_v, ext\_U_v] \supseteq [T_{x,v} - d(sp, p_{x,v}) \,,\, T_{x,v} + d(p_{x,v}, sp)] \,,$$

and the lemma is proven in this case.

To complete the proof, consider the case that either $d_{x,v}(sp, q) = \infty$ or $d_{x,v}(p_{x,v}, sp) = \infty$. Suppose, for example, that $d_{x,v}(sp, p_{x,v}) = \infty$ (the other case is analogous). In this case we apply Theorem 5.8 and get that for any $N > 0$ there exists an execution e_N with view $\mathcal{V}$ in which $\delta(p_{x,v}, sp) > N$. Therefore, in e_N, when the local time at v is $T_{x,v}$, the source time is greater than $T_{x,v} + N$. Since Theorem 5.7 also says that the output of the CSA at v is identical for all e_N, the correctness requirement implies that in x, $ext\_L_v = -\infty$. ∎

The following theorem shows that the lower bound on tightness of Theorem 6.3 is an upper bound too.

Theorem 6.4 *There exists an external CSA such that for any state x in an execution of the clock synchronization system, at any processor v, the output values are*

$$\begin{aligned} ext\_L_v &= T_{x,v} - d_{x,v}(sp, p_{x,v}) \\ ext\_U_v &= T_{x,v} + d_{x,v}(p_{x,v}, sp) \,. \end{aligned}$$

Proof Sketch: The proof consists of the specification of the algorithm. Below, we outline a simple algorithm, based on the full information protocol. More specifically, the state of the CSA at a processor v describes the complete local view of v at that state. Using the standard bounds mapping (assumed to be built into the algorithm), the synchronization graph can be computed, and the output values are given by

$$ext\_L_v = local\_time_v - d_{x,v}(sp, p_{x,v}) \tag{6.2}$$

$$ext\_U_v = local\_time_v + d_{x,v}(p_{x,v}, sp) \,. \tag{6.3}$$

The implementation of the algorithm is straightforward: a description of the complete current local view (where each point has a unique name) is sent in every message; whenever a message arrives, the view it carries is merged in the natural way with the current local view by performing union over the two graphs. A synchronization graph is then constructed from the new view and its standard bounds mapping, and the distances from the current point to the source point and from the source point to the current point are computed, using any single-source shortest paths algorithm for general graphs (see, e.g., [5]). Using these distances, the output variables are updated according to Eqs. (6.2, 6.3). To have updated output values at all states, the output variables are also modified whenever a time-passage action occurs: if the local time is incremented by b units, we set

$$
\begin{aligned}
ext\_L_v &\leftarrow ext\_L_v + b(\overline{\varrho}_v - 1)/\overline{\varrho}_v && (6.4)\\
ext\_U_v &\leftarrow ext\_L_v + b(1 - \underline{\varrho}_v)/\underline{\varrho}_v\ . && (6.5)
\end{aligned}
$$

This completes the description of the algorithm. Let us now explain why is it correct. First, we argue that the algorithm describes admissible CSA modules: it has the required interface, it has the non-interfering filtering property, it is real-time blind, and its initial states are quiescent. To show correctness, we apply an easy induction on the steps of the execution that shows that the algorithms maintains, at each point, a description of the local view from that point, and therefore the output is correct after each receive event. Consider now the synchronization graph at the null point $p_{x,v}$ that occurs at v at local time $T_{x,v}$. Let p'_v be the last receive point that occurs at v before $p_{x,v}$. If p'_v does not exist, we are done trivially, since both the synchronization distances and the output values are infinite in this case. Otherwise, by the definitions we get that there is a single path from $p_{x,v}$ to p'_v with weight $virt\_del(p_{x,v}, p'_v)(1 - \underline{\varrho}_v)/\underline{\varrho}_v$. Similarly, there exists a single path from p'_v to $p_{x,v}$, with weight $virt\_del(p'_v, p_{x,v})(\overline{\varrho}_v - 1)/\overline{\varrho}_v$. Hence, from Eqs. (6.2–6.5) and Lemma 6.2, we have that the algorithm is correct. Finally, note that the output values satisfy the theorem statement, by the specification of the algorithm and by the fact that its state at any point represents the local view at that point. ∎

Remarks.

1. The algorithm above is *optimal*, as defined in Definition 4.1, i.e., it provides the best possible output values at each point.

2. It is easy to make the algorithm described above more efficient without affecting the output. For example, instead of sending the complete view in each message, it suffices to

send only incremental changes. Notice that this modification would reduce the communication overhead significantly, but would not help to save space for storing state (in fact, more space will be needed at the processors). The property of high space requirement is inherent to optimal algorithms for general systems, as we show in Chapter 8.

6.3 An Efficient Algorithm for Drift-Free Clocks

In this section we restrict our attention to the case where all clocks are drift-free. Making this simplifying assumption enables us to derive an extremely efficient algorithm for external synchronization that gives optimal tightness. The algorithm is presented in Subsection 6.3.1, and analyzed in Subsection 6.3.2.

6.3.1 The Algorithm

The complete specification of the algorithm given in Figure 6-1 (non-source processors) and Figure 6-2 (source processors). The code lines that are not part of the generic code for CSAs are numbered. The idea is as follows. As proved in Lemma 6.1, all the points that occur at a processor with a drift-free clock can be thought of as a single superpoint for distance computations. Intuitively, our algorithm computes distances in the graph of superpoints. Since arc weights in the graph of superpoints may only decrease, we use (two independent versions of) the distributed Bellman-Ford algorithm for single-source shortest paths computation [4].

More specifically, for each link L_{uv}, the CSA at node v maintains estimates for the weight of the lightest arcs from the superpoint of u to v in the state variable $\tilde{\mathtt{w}}(u,v)$, and of weight of the lightest arcs from v to u in state variable $\tilde{\mathtt{w}}(v,u)$. To this end, whenever a message arrives, the weight of the corresponding arcs in the synchronization graph are computed, using a temporary variable $\tilde{\mathtt{v}}$ which holds the virtual delay, and the message latency bounds; only the minimum estimate is kept (lines 4-6 and 5s-7s). Using these weights, the distances to and from the source are computed in the variables $\tilde{\mathtt{d}}(v,s)$ and $\tilde{\mathtt{d}}(s,v)$, respectively. Lines 7-8 in are the Bellman-Ford relaxations. In lines 9-10, the output variables are updated.

In addition, whenever a message is sent to a neighbor, the CSA augments it with the current local time, the best known weights for the arcs between them, and the distances to and from the source (lines 3 and 4s).

The problem specification also requires that the output variables be updated when time passes (lines 11-12).

6.3.2 Correctness and Optimality

We now prove that the algorithm above is an optimal external CSA. First we state the following easy fact.

Lemma 6.5 *The algorithm in Figures 6-1 and 6-2 is an admissible CSA.*

Proof: We verify the following according to Definition 3.2.

- Clearly, the algorithm has the interface as in Figure 3-5.
- It is straightforward to see that the algorithm has the non-interfering filtering property: the code is based on the generic CSA of Figure 3-6.
- It is also easy to see that the algorithm is real-time blind, since the transitions never refer to the *now* component of the state (lines 11-12 are based on the difference in local times).
- Finally, the initial states of the algorithm above are quiescent: no internal or output actions are enabled an in initial state, nor in any state reachable by time passage from them. ∎

We now turn to the less obvious part, namely proving that the algorithm above is an optimal external CSA. Before we start, we introduce the following notion.

Definition 6.1 *Let u, v be two neighbor processors in a clock synchronization system. Given a synchronization graph $\Gamma = (V, E, w)$, the set $W^{uv}(\Gamma)$ is defined to be the set of all numbers $w(p, q)$, where p occurs at u, q occurs at v, and $(p, q) \in E$.*

The key for the optimality of the algorithm is the following lemma.

Lemma 6.6 *Let p be a point in an execution of the system above, and suppose that p occurs at processor v. Let $\Gamma = (V, E, w)$ be the synchronization graph generated by the local view of the execution at p and its standard bounds mapping. Let $\tilde{\mathtt{w}}$ and $\tilde{\mathtt{d}}$ denote the value of the local variables of v at in the state following p. Then the following invariant holds.*

(1) For all neighbors u of v, $\tilde{\mathtt{w}}(v, u) = \min(W^{vu}(\Gamma))$ and $\tilde{\mathtt{w}}(u, v) = \min(W^{uv}(\Gamma))$.

Sites: *a single non-source site* v

State

now: *non-negative real number, initially* 0
local_time: *real number, initially arbitrary*
ext_L: *real number, initially* $-\infty$
ext_U: *real number, initially* ∞
Q_i: *queue for symbols of* Σ, *initially* $\emptyset$
Q_o: *queue for symbols of* $\Sigma \times \mathbf{R}^5$, *initially* $\emptyset$
active: *Boolean flag, initially* FALSE
$\tilde{\mathtt{d}}_v(v,s), \tilde{\mathtt{d}}_v(s,v)$: *real numbers, initially* ∞ 1
$\tilde{\mathtt{w}}(v,u)$ *and* $\tilde{\mathtt{w}}_v(u,v)$ *for each* $u \in \mathcal{N}(v)$: *real numbers, initially* ∞ 2

Actions

$Send\_Message_v^u(m)$ *(input)*
Eff: *enqueue* m *in* Q_o
$active \leftarrow$ TRUE

$Send\_Aug\_Message_v^u(m_1, m_2)$ *(output)*
Pre: m_1 *is at the head of* Q_o
$m_2 = \langle local\_time, \tilde{\mathtt{w}}(v,u), \tilde{\mathtt{w}}(u,v), \tilde{\mathtt{d}}(v,s), \tilde{\mathtt{d}}_u(s,v) \rangle$ 3
Eff: *remove head of* Q_o
if $Q_o = Q_i = \emptyset$ **then** $active \leftarrow$ FALSE

$Receive\_Aug\_Message_v^u(m_1, \langle local\_time_u, \tilde{\mathtt{w}}_u(v,u), \tilde{\mathtt{w}}_u(u,v), \tilde{\mathtt{d}}_u(s,u), \tilde{\mathtt{d}}_u(u,s) \rangle)$ *(input)*
Eff: *enqueue* m_1 *in* Q_i
$active \leftarrow$ TRUE
$\tilde{\mathtt{v}} \leftarrow local\_time - local\_time_u$ 4
$\tilde{\mathtt{w}}(v,u) \leftarrow \min\{H(m_1) - \tilde{\mathtt{v}}\ ,\ \tilde{\mathtt{w}}_u(v,u)\ ,\ \tilde{\mathtt{w}}(v,u)\}$ 5
$\tilde{\mathtt{w}}(u,v) \leftarrow \min\{-L(m_1) + \tilde{\mathtt{v}}\ ,\ \tilde{\mathtt{w}}_u(u,v)\ ,\ \tilde{\mathtt{w}}(u,v)\}$ 6
$\tilde{\mathtt{d}}(v,s) \leftarrow \min\left\{\tilde{\mathtt{w}}(v,u) + \tilde{\mathtt{d}}_u(u,s)\ ,\ \tilde{\mathtt{d}}(v,s)\right\}$ 7
$\tilde{\mathtt{d}}(s,v) \leftarrow \min\left\{\tilde{\mathtt{d}}_u(s,u) + \tilde{\mathtt{w}}(u,v)\ ,\ \tilde{\mathtt{d}}(s,v)\right\}$ 8
$ext\_L \leftarrow local\_time - \tilde{\mathtt{d}}(s,v)$ 9
$ext\_U \leftarrow local\_time + \tilde{\mathtt{d}}(v,s)$ 10

$Receive\_Message_v^u(m_1)$ *(output)*
Pre: m_1 *is at the head of* Q_i
Eff: *remove head of* Q_i
if $Q_o = Q_i = \emptyset$ **then** $active \leftarrow$ FALSE

ν : *(time passage)*
Pre: $active =$ FALSE
$b > 0$
Eff: $now \leftarrow now + b$
$local\_time \leftarrow local\_time + b$ 11
$ext\_L \leftarrow ext\_L + b$ 12
$ext\_U \leftarrow ext\_U + b$

Figure 6-1: *Code of optimal CSA protocol for external synchronization with drift-free clocks: a non-source processor. The non-generic code lines are numbered.*

Sites: *the source site* s

State

now: *non-negative real number, initially* 0
local_time: *real number, initially arbitrary*
ext_L, *ext_U*: *real number, always equal to local_time* 1s
Q_i: *queue for symbols of* Σ, *initially* $\emptyset$
Q_o: *queue for symbols of* $\Sigma \times \mathbf{R}^5$, *initially* $\emptyset$
active: *Boolean flag, initially* FALSE
$\tilde{\mathtt{d}}_v(s,s), \tilde{\mathtt{d}}_v(s,s)$: *always* 0 2s
$\tilde{\mathtt{w}}(s,u)$ *and* $\tilde{\mathtt{w}}(u,s)$ *for each* $u \in \mathcal{N}(s)$: *real numbers, initially* ∞ 3s

Actions

Send_Message$_s^u(m)$ *(input)*
  Eff: *enqueue* m *in* Q_o
    active $\leftarrow$ TRUE

Send_Aug_Message$_s^u(m_1, m_2)$ *(output)*
  Pre: m_1 *is at the head of* Q_o
    $m_2 = \langle local\_time, \tilde{\mathtt{w}}(s,u), \tilde{\mathtt{w}}(u,s), 0, 0\rangle$ 4s
  Eff: *remove head of* Q_o
    if $Q_o = Q_i = \emptyset$ **then** *active* $\leftarrow$ FALSE

Receive_Aug_Message$_s^u(m_1, \langle local\_time_u, \tilde{\mathtt{w}}_u(s,u), \tilde{\mathtt{w}}_u(u,s), \tilde{\mathtt{d}}_u(s,u), \tilde{\mathtt{d}}_u(u,s)\rangle)$ *(input)*
  Eff: *enqueue* m_1 *in* Q_i
    active $\leftarrow$ TRUE
    $\tilde{\mathtt{v}} \leftarrow local\_time - local\_time_u$ 5s
    $\tilde{\mathtt{w}}(s,u) \leftarrow \min\{H(m_1) - \tilde{\mathtt{v}}\,,\, \tilde{\mathtt{w}}_u(s,u)\,,\, \tilde{\mathtt{w}}(s,u)\}$ 6s
    $\tilde{\mathtt{w}}(u,s) \leftarrow \min\{-L(m_1) + \tilde{\mathtt{v}}\,,\, \tilde{\mathtt{w}}_u(u,s)\,,\, \tilde{\mathtt{w}}(u,s)\}$ 7s

Receive_Message$_s^u(m_1)$ *(output)*
  Pre: m_1 *is at the head of* Q_i
  Eff: *remove head of* Q_i
    if $Q_o = Q_i = \emptyset$ **then** *active* $\leftarrow$ FALSE

ν : *(time passage)*
  Pre: *active* = FALSE
    $b > 0$
  Eff: *now* $\leftarrow$ *now* $+ b$
    local_time $\leftarrow$ *local_time* $+ b$

Figure 6-2: *Code of optimal CSA protocol for external synchronization with drift-free clocks: a source processor. The non-generic code lines are numbered.*

(2) Let sp be the source point of Γ*. Then* $d_\Gamma(sp, p) = \tilde{\mathtt{d}}(s, v)$*, and* $d_\Gamma(p, sp) = \tilde{\mathtt{d}}(v, s)$*.*

Proof: The lemma is proven by induction on the steps of e, with the initial state as a base case. For the base case, we observe that the invariant holds for all processors in the initial states of the system by lines 1-2 and 2s-3s of the code, since Γ is empty then.

For the inductive step, let p' be the last event at v before p, or the initial state if no such event exists. If p' is a point, let $\Gamma' = (V', E', w')$ be the synchronization graph generated by the local view of the execution at p' and its standard bounds mapping, and otherwise define Γ' to be the empty graph. To prove the inductive step, we consider two cases.

Case 1: p is a send event. In this case, by Def. 5.4, $V = V' \cup \{p\}$, and if Γ' is not empty, then $E = E' \cup \{(p, p'), (p', p)\}$, $w(e') = w'(e')$ for all $e' \in E'$, and by Def. 3.11, $w(p, p') = w(p', p) = 0$. By the inductive hypothesis, the invariant holds at p'. Hence, $W^{vu}(\Gamma) = W^{vu}(\Gamma')$ and $W^{uv}(\Gamma) = W^{uv}(\Gamma')$. Since by the code, the $\tilde{\mathtt{w}}$ variables are unchanged by a send event, we have that part (1) of the invariant holds in p. For part (2), note that there is only one arc incoming into p, and one arc outgoing from p. Since both arcs have weight 0, and since they connect p to p', it follows that $d_\Gamma(p, p_0) = d_{\Gamma'}(p, p_0)$, and that $d_\Gamma(p_0, p) = d_{\Gamma'}(p_0, p)$. Again, since the algorithm does not change the value of the $\tilde{\mathtt{d}}$ variables when a send event occurs, part (2) of the invariant holds in this case.

Case 2: p is a receive event. Specifically, assume that p is the following event:

$$\mathit{Receive\_Aug\_Message}_v^u(m_1, \langle \mathit{local\_time}, \tilde{\mathtt{w}}(v, u), \tilde{\mathtt{w}}(u, v), \tilde{\mathtt{d}}(v, s), \tilde{\mathtt{d}}_u(s, v)\rangle)$$

Denote the corresponding send event at u by p'', and let $\Gamma'' = (V'', E'', w'')$ be the synchronization graph generated by the local view at p'' and the standard bounds mapping. By definitions, $V = V' \cup V'' \cup \{p\}$, and either $E = E' \cup E'' \cup \{(p, p''), (p'', p)\}$ if Γ' is empty, or $E = E' \cup E'' \cup \{(p, p''), (p'', p), (p, p'), (p', p)\}$ if Γ' is not empty. The weights are defined by

$$w(e) = \begin{cases} w'(e), & \text{if } e \in E' \\ w''(e), & \text{if } e \in E'' \\ H(m_1) - \mathit{virt\_del}(p, p''), & \text{if } e = (p, p'') \\ -L(m_1) - \mathit{virt\_del}(p'', p), & \text{if } e = (p'', p) \\ 0, & \text{if } e \in \{(p, p'), (p', p)\} \end{cases}$$

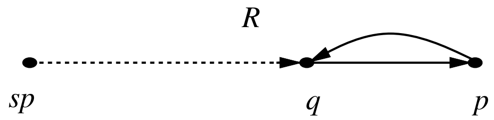

Figure 6-3: *Scenario considered in the proof of Lemma 6.6. R is a shortest path from sp to p with last arc* (q, p).

Part (1) of the invariant in this case is proven as follows. By definitions, $W^{uv}(\Gamma) = W^{uv}(\Gamma') \cup W^{uv}(\Gamma'') \cup \{w(p, p'')\}$, and $W^{vu}(\Gamma) = W^{vu}(\Gamma') \cup W^{vu}(\Gamma'') \cup \{w(p'', p)\}$. Hence

$$\min(W^{vu}(\Gamma)) = \min\Big(W^{vu}(\Gamma') \cup W^{vu}(\Gamma'') \cup \{H(m_1) - \mathit{virt\_del}(p, p'')\}\Big) ,$$

and

$$\min(W^{uv}(\Gamma)) = \min\Big(W^{uv}(\Gamma') \cup W^{uv}(\Gamma'') \cup \{-L(m_1) - \mathit{virt\_del}(p'', p)\}\Big) ,$$

which, according to the inductive hypothesis applied to p' and p'', is exactly the calculation in lines 4-6 and 5s-7s. This proves part (1) of the invariant.

For the second part of the invariant, let us prove that $\tilde{\mathtt{d}}(s, v) = d_\Gamma(sp, p)$. The claim is trivial for $v = s$, according to line 2s. So suppose $v \neq s$. Consider a shortest path from sp to p that contains no cycles. This is possible since by Lemma 5.3, all cycles in Γ have non-negative weight. Focus on the last arc of the path in question, i.e., the arc that leads to p (see Figure 6-3). Denote this arc (q, p), where $q \in \{p', p''\}$, and let Γ^* be the synchronization graph at q. By the choice of q, $d_\Gamma(sp, p) = d_\Gamma(sp, q) + w(q, p)$. By the induction hypothesis, we have that at q, the $\tilde{\mathtt{d}}$ variables are equal to the corresponding distances in Γ^*. Also, we have that after line 7, $\tilde{\mathtt{w}}(v, u) = \min(W^{vu}(\Gamma))$ and $\tilde{\mathtt{w}}(u, v) = \min(W^{uv}(\Gamma))$. Therefore, by Line 9 of the code, it suffices to prove that $d_\Gamma(sp, q) = d_{\Gamma^*}(sp, q)$. We do this in two steps. First, notice that $d_\Gamma(sp, q) \leq d_{\Gamma^*}(sp, q)$ since Γ^* is a subgraph of Γ. Next we argue that $d_\Gamma(sp, q) \geq d_{\Gamma^*}(sp, q)$ by contradiction: suppose that $d_\Gamma(sp, q) < d_{\Gamma^*}(sp, q)$. Then all shortest paths from sp to q in Γ are shorter than the shortest path from sp to q in Γ^*. Consider such a shortest path which is simple (this is possible since Γ has no negative-weight cycles). This path must end with the arc (p, q), or otherwise it is completely contained in Γ^*. It follows that the shortest path from sp to p goes through p, q, and back to p (see Figure 6-3), a contradiction to the choice of the path as simple. Therefore, $d_\Gamma(sp, q) \geq d_{\Gamma^*}(sp, q)$, and we conclude that $d_\Gamma(sp, q) = d_{\Gamma^*}(sp, q)$.

To show that $\tilde{\mathtt{d}}(v,s) = d_\Gamma(p, sp)$, we repeat the symmetrical argument for the first arc of a simple shortest path from p to sp, and use line 8 of the code instead of line 9. ▮

We can now prove the optimality of the algorithm.

Theorem 6.7 *The CSA algorithm in Figure 6-1 and Figure 6-2 is an optimal algorithm (in the sense of Def. 4.1) for all external synchronization environments, where all clocks are drift-free.*

Proof: Clearly, the algorithm may be composed with any environment of external synchronization, where all clocks are drift-free. Consider any state x of an execution of the algorithm, let v be any processor, and let $T_{x,v} = local\_time_v(x)$. Let Γ be the synchronization graph generated by the local view of v at time $T_{x,v}$ and the standard bounds mapping. Denote the null point in Γ that occurs at v at local time $T_{x,v}$ by $p_{x,v}$. Let p' be the last point that occurs at v before $p_{x,v}$, and let Γ' be the synchronization graph generated by the local view at p' and the standard bounds mapping. By Lemma 6.1, $d_\Gamma(p_{x,v}, sp) = d_{\Gamma'}(p', sp)$, and $d_\Gamma(sp, p_{x,v}) = d_{\Gamma'}(sp, p')$. Hence

$$\begin{array}{lcll} source\_time(x) & \in & [T_{x,v} - d_\Gamma(sp, p_{x,v}), T_{x,v} + d_\Gamma(p_{x,v}, sp)] & \textit{by Lemma 6.2} \\ & = & [ext\_L, ext\_U] & \textit{by lines 9-12 and Lemma 6.6} \end{array}$$

This means that the algorithm is correct. The optimality of the algorithm follows immediately from the lower bound of Theorem 6.3. ▮

6.4 The Round-Trip Technique

It may be interesting at this point to compare our analysis and algorithms with the common clock synchronization technique known as "round-trip probes." For concreteness, we take the external synchronization system NTP (Network Time Protocol, the clock synchronization algorithm used over the Internet [26]) as our prime source for this technique. We consider here a simplified variant of NTP, called SNTP, that was introduced in Section 3.1.5. In the SNTP system, we have only two processors with drift-free clocks, connected by perfect asynchronous links. We denote the source processor by s, and the non-source processor by v. SNTP is rigorously defined in Section 3.1.5, with a technique for a single

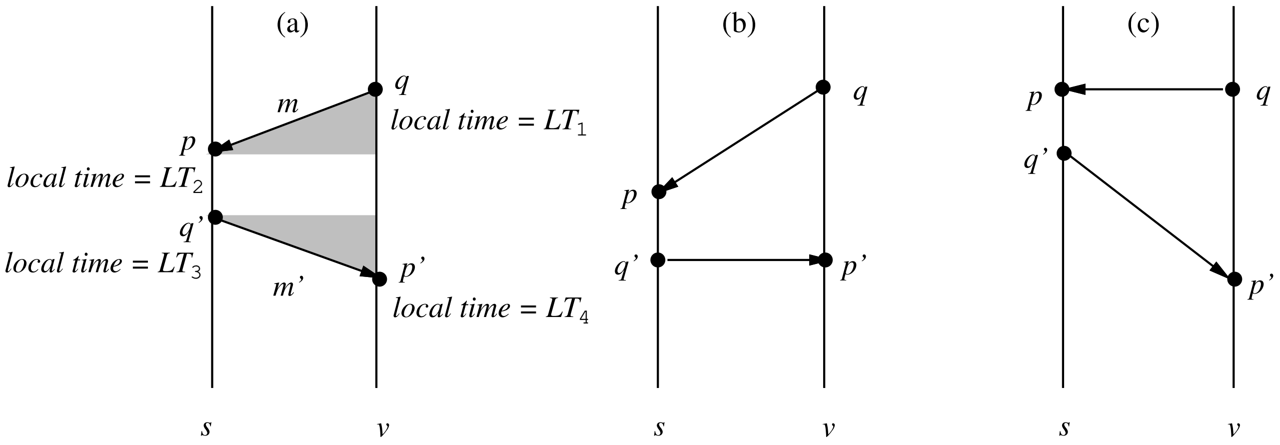

Figure 6-4: *Reproduction of Fig. 3-7. (a) A typical round trip technique. (b) m is in transit TT time units. (c) m' is in transit TT time units.*

round trip. In this section, we extend the presentation to multiple round-trips, and focus on the way their results are combined. Let us recall briefly the main ideas.

Periodically, v sends a message to s, which in turn responds by sending a message back to v (hence the name "round trip"). Consider the round trip depicted in Figure 6-4(a), where v sends a message m to s, and s responds by sending m' to v. Let TT denote the total transit time of m and m'. The bounds on the source time are obtained by considering two extreme scenarios, in which on message is in transit TT time units and the other is delivered instantaneously (Figure 6-4 (b,c)). Skipping the details (they can be found in Section 3.1.5), we remark that the bounds generated by the CSA module at v at point q' are

$$[ext\_L\ ,\ ext\_U] = [LT_3\ ,\ LT_3 + TT]\ .$$

Clearly, the tightness of the synchronization thus computed is exactly the total transit time. In other words, the faster the messages are delivered, the better synchronization is achieved. This fact leads the designers of NTP to the following conclusion: when there are many round trips, the one with the least total transit time is chosen as best, and its corresponding bounds are output. Specifically, whenever a round trip is completed, its total transit time is compared against the current tightness; if the current tightness is better (i.e., smaller), that round trip is discarded, and otherwise, the bounds obtained by that round-trip replace the current values of the output variables. The formal specification of the CSA at v for multiple round-trips is given in Figure 6-5 (note the "if then" clause in the effect of the *Receive_Aug_Message* action). The code for the source processor is identical to the case of a single round-trip (see Figure 3-9).

Let us now consider the behavior of the algorithm described in Section 6.3 for this toy environment. Note that the patterns generated by the environment of SNTP are a subset of the patterns generated by the general environment described in Section 3.1, and therefore it makes sense to consider the CSAs of Section 6.3 in the context of the environment of SNTP.

Our first remark regards the single round-trip scenario depicted in Figure 6-4 (a). Using Definitions 3.11 and 5.4, we get that the synchronization graph corresponding to this scenario is the one depicted in Figure 6-6. It is straightforward to verify that the extreme scenarios depicted in Figure 3-7 (b,c) are, in fact, the executions whose existence is guaranteed by Theorem 5.8 for this view and bounds mapping. As a consequence, the output of the algorithm of Section 6.3, and the bounds computed by SNTP are identical in this case.

However, in a scenario that consists of more than a single round-trip, the algorithm of Section 6.3 may do much better. By computing the distances in the synchronization graph, our algorithm in effect finds the fastest message delivered over the link in each direction *independently*, while SNTP finds the best *round-trip* using a pre-specified matching of the messages into pairs.

Let us consider a concrete example. In Figure 6-7 (a) we have a diagram of a two-round-trip scenario. Suppose that the total transit time of the first round-trip is smaller than the one in the second, i.e., let $TT_1 = (LT_4 - LT_1) - (LT_3 - LT_2)$, let $TT_2 = (LT_8 - LT_5) - (LT_7 - LT_6)$, and assume $TT_1 < TT_2$. In this case, the tightness of synchronization produced by SNTP after the scenario is TT_2. By contrast, the algorithm of Section 6.3 finds the best *possible* round trip in the execution: in our example, the picture suggests that $TT^* = (LT_8 - LT_1) - (LT_7 - LT_2)$ is the best choice, and in particular, $TT^* < TT_1$. Notice that TT^* may be arbitrarily smaller than TT_1, and hence the local competitive factor of SNTP cannot be bounded even in this simple case.

Intuitively, the round-trip technique used by NTP is handicapped since it potentially pairs a "good" message in one direction with a "bad" message in the other direction. We remark that in the case of a system of more than one link, the pairing of good and bad messages may be even more severe: consider the set of messages used to establish the bounds of the output variables. These messages correspond to paths (in the synchronization graph)

Sites: a single site v

State

now: non-negative real number, initially 0
local_time: real number, initially arbitrary
ext_L: real number, initially $-\infty$
ext_U: real number, initially ∞
Q_i*: queue for symbols of* Σ*, initially* $\emptyset$
Q_o*: queue for symbols of* $\Sigma \times \mathbf{R}^2$*, initially* $\emptyset$
active: Boolean flag, initially FALSE
LT_1*: a real number, initially undefined*

Actions

$Send\_Message_v(m)$ *(input)*
Eff: $enqueue\ m\ in\ Q_o$
$active \leftarrow \text{TRUE}$
$LT_1 \leftarrow local\_time$

$Send\_Aug\_Message_v(m_1, 0, 0)$ *(output)*
Pre: m_1 *is at the head of* Q_o
Eff: *remove head of* Q_o
if $Q_o = Q_i = \emptyset$ **then** $active \leftarrow \text{FALSE}$

$Receive\_Aug\_Message_v(m_1, \langle LT_2, LT_3 \rangle)$ *(input)*
Eff: *enqueue* m_1 *in* Q_i
$active \leftarrow \text{TRUE}$
$LT_4 \leftarrow local\_time$
$TT \leftarrow (LT_4 - LT_1) - (LT_3 - LT_2)$
if $TT < (ext\_U - ext\_L)$ **then**
$ext\_L \leftarrow LT_3$
$ext\_U \leftarrow LT_3 + TT$

$Receive\_Message_v(m_1)$ *(output)*
Pre: m_1 *is at the head of* Q_i
Eff: *remove head of* Q_i
if $Q_o = Q_i = \emptyset$ **then** $active \leftarrow \text{FALSE}$

ν : *(time passage)*
Pre: $active = \text{FALSE}$
$b > 0$
Eff: $now \leftarrow now + b$
$local\_time \leftarrow local\_time + b$
$ext\_L \leftarrow ext\_L + b$
$ext\_U \leftarrow ext\_U + b$

Figure 6-5: *Code of the CSA module in SNTP for processor v (the best round-trip is chosen).*

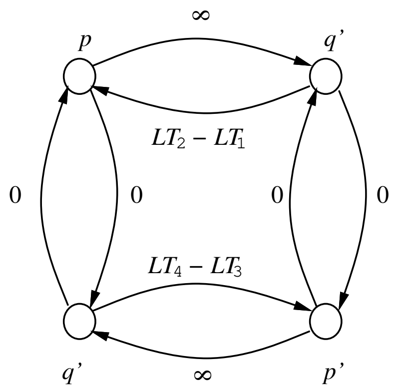

Figure 6-6: *The synchronization graph corresponding to the scenario in Fig. 6-4 (a), assuming that the clocks are drift-free and that transmission time of the messages are can be any value between 0 and ∞.*

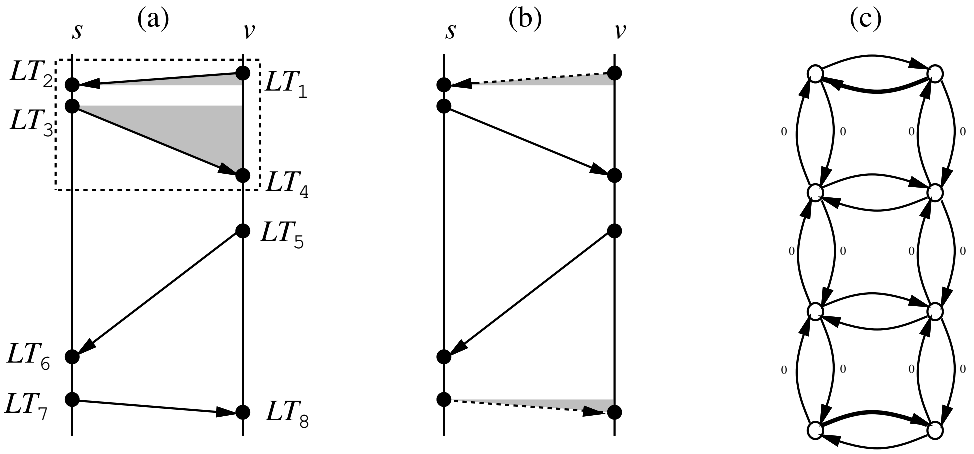

Figure 6-7: *A time space diagram of two round trips is given in (a), with local times of the points. SNTP chooses the round trip with the smallest total transit time (enclosed in the dashed frame in (a)). For the same scenario, the algorithm of Section 6.3 implicitly chooses the best message in each direction independently, and in effect finds the best possible round trip (dashed arrows in (b)). The corresponding synchronization graph is given in (c), where the lightest arcs connecting points of s and v are boldfaced.*

to and from the source. The round trip technique forces both paths to be over the same physical links, i.e., the messages used in one direction must be transmitted over the same links over which the messages used in the other direction were transmitted. Our algorithm, by contrast, chooses messages independently for each direction, and it may well be the case that the set of messages used to establish a lower bound are transmitted over different links over which the messages used for the upper bound were transmitted.

Summary

In this chapter we defined and analyzed the external clock synchronization problem. In this problem, a distinguished *source* processor is assumed to have a drift-free clock, and the task of all processors is to keep updated bounds on the current value of the source clock. Using synchronization graphs, we derived matching lower and upper bounds on external synchronization in general systems, where the clocks of non-source processors may have arbitrary drift bounds and messages may have arbitrary latency bounds.

The algorithm used for the upper bound is a full information protocol, and therefore it is inefficient. By contrast, we presented an extremely efficient algorithm for the case of drift-free clocks. The latter algorithm is based on the observation that all points associated with a drift free clock in the synchronization graph can be collapsed into a single superpoint, and thus it is sufficient to compute distances between superpoints.

We have also examined the popular technique of round trips. Using a toy system based on NTP, we showed that for a single round trip this technique yields the same result as our algorithm. In a multiple round-trip scenario, however, the output of our algorithm will be usually better.

Chapter 7

Internal Synchronization

In this chapter we prove a lower bound on the tightness of another variant of clock synchronization, called *internal clock synchronization* [6]. The goal of internal synchronization is that all processors generate a "tick," called *fire* below, such that all *fire* steps occur in the smallest possible interval of real time. An algorithm for internal synchronization is required to provide bounds on the length of this real time interval, and the smallest difference in an execution is the *internal tightness* of that execution.

The task of internal synchronization has been the target of considerable research (see, e.g., [19, 7, 13, 3] and the survey [31]). However, to the best of our knowledge, the only known non-trivial lower bounds for internal tightness were for the case of drift-free clocks. In this chapter, based on synchronization graphs, we give a lower bound for the internal tightness in a synchronization system with bounded-drift clocks. We remark that the lower bound presented in this chapter is based on views, rather than local views: lower bounds that hold for a given view hold *a fortiori* for its local views.

This chapter is organized as follows. In Section 7.1 we define internal clock synchronization formally, and in Section 7.2 we present the lower bound.

7.1 Definition of Internal Synchronization

In this section we recall our definition of internal synchronization (see Section 4.1). An internal clock synchronization system is a clock synchronization system, where each CSA module has a special internal action called *fire*.[1] The correctness requirement of the internal

[1] The *fire* action is internal so as to keep the interface of CSAs standard (see Figure 3-5).

synchronization task is that

(1) each processor v takes a $fire_v$ action exactly once during an execution of the system, and

(2) the CSA at each processor v maintains output variables called int_L_v and int_U_v, such that at all states, the real time interval $[now(fire_v) + int\_L_v, now(fire_v) + int\_U_v]$ contains all the *fire* events in the execution.

The *internal tightness* of an execution of an internal synchronization system at a processor v, denoted $tightness_v(e)$, is the infimum over the difference $(int\_U_v - int\_L_v)$ in all states of the execution.

Intuitively, the *fire* actions represent the event of resetting some logical clock maintained by the CSAs; the output variables express the synchronization guarantee made by the CSA. By the properties of CSAs (specifically, their real-time blindness and their quiescent initial states), one can show that their initial values must be $int\_L = -\infty$ and $int\_U = \infty$; as the execution progresses, the CSA modules gather information about the occurrence of remote *fire* actions that may enable them to reduce the difference between their output values.

7.1.1 Discussion

Intuitively, the motivation for internal synchronization is to maintain some clock variables in each processor, such that their values are as close as possible. This requirement alone is not sufficient, since it allows for the trivial solution where all clock variables always have the same fixed value (say, 0). Dolev *et al.* discuss this issue in depth [7]. In [19], this difficulty is avoided as follows. Each processor v is assumed to have a special output variable denoted CORR_v; the tightness is measured as the maximal difference between the values of $local\_time_v + \mathrm{CORR}_v$, over all processors v. To rule out the trivial solution of setting $\mathrm{CORR}_v = -local\_time_v$, in [19] the executions of synchronization algorithms are required to be finite, i.e., at some point the algorithm enters a terminating state, after which the CORR variable is fixed. The tightness is defined to be the maximal difference between the $local\_time_v + \mathrm{CORR}_v$ values, measured only when the algorithm is in a final state.

In [13], the difficulty of problem definition is solved differently: each processor is required to flip a special internal bit during the execution of the algorithm; the tightness is defined to be the maximal difference in real time between two remote bit flips. We adopted this definition (the bit flip is equivalent to our *fire* action), and added the output variables for ease of exposition.

7.2 A Lower Bound on Internal Tightness

In this section we derive a lower bound on the tightness of internal synchronization in general systems with bounded-drift clocks. To state the result, we define the following graph-theoretic concept. Recall that for a path θ in a weighted graph, $w(\theta)$ denotes the sum of the weights of arcs in θ, and let $|\theta|$ denote the number of arcs in θ.

Definition 7.1 *Let $G = (V, E, w)$ be a weighted directed graph. The* maximum cycle mean *of G, denoted* $\mathrm{mcm}(G)$*, is the maximum average weight of an edge in a directed cycle of G. That is,* $\mathrm{mcm}(G) = \max\{w(\theta)/|\theta| \ : \ \theta \text{ is a directed cycle of } G\}$.

We remark that the maximum cycle mean can be computed in polynomial time [14].

To analyze internal synchronization systems, the definition of patterns and views is extended so that the *fire* steps are points with the usual attributes (i.e., processor of occurrence, local time of occurrence, and for patters, real time of occurrence). We extend the standard bounds mapping too, using Def. 3.11. Synchronization graphs for internal synchronization systems are thus also naturally defined. It turns out that the following derivative of synchronization graphs is useful for the analysis of internal synchronization.

Definition 7.2 *Given a synchronization graph $\Gamma = (V, E, w)$ of an internal clock synchronization system, the* internal synchronization graph *is a directed, weighted graph $\overline{\Gamma} = (\overline{V}, \overline{E}, \overline{w})$, where the set of points $\overline{V}$ consists of all the fire points in V; there is an arc in $\overline{E}$ between every pair of points of $\overline{V}$; and $\overline{w}(\mathit{fire}_v, \mathit{fire}_u) = d_\Gamma(\mathit{fire}_v, \mathit{fire}_u)$ for each $(\mathit{fire}_v, \mathit{fire}_u) \in \overline{E}$.*

We can now state and prove the lower bound.

Theorem 7.1 *Let e be an execution of an internal clock synchronization system, and let $\overline{\Gamma}$ be the internal synchronization graph generated by the view of e and the standard bounds mapping. Then $\mathit{tightness}_v(e) \geq \mathrm{mcm}(\overline{\Gamma})$ for all processors v.*

Proof: Suppose first that $\mathrm{mcm}(\overline{\Gamma}) = \infty$. Then, by the definition of $\overline{\Gamma}$, there are some processors u, v with $d_\Gamma(\mathit{fire}_v, \mathit{fire}_u) = \infty$. Hence, by Theorem 5.7, for any $N > 0$ there exists an execution e_N, in which $\delta(\mathit{fire}_v, \mathit{fire}_u) > N$. Moreover, since the output variables are part of the basic component of the state of CSAs, we have from Theorem 5.7 that the set of output values of the CSA at v are identical in all the e_N. Let act_del_N denote the actual delay function in e_N. Since for any two points in any execution we have $\delta(p,q) = act\_del(p,q) - virt\_del(p,q)$, and since $virt\_del(\mathit{fire}_v, \mathit{fire}_u)$ is fixed (it is a part of the view of e), it follows that the set of numbers $\{act\_del_{e_N}(\mathit{fire}_v, \mathit{fire}_u) : N > 0\}$ cannot be bounded. Therefore, by the correctness requirement for internal CSAs, we must have $tightness_v(e) = \infty$ for all processors v, and the theorem holds in this case.

Consider now the case where $\mathrm{mcm}(\overline{\Gamma}) < \infty$. Let $\theta = \langle p_0, p_1, \ldots, p_{|\theta|} = p_0 \rangle$ be an arbitrary directed cycle in $\overline{\Gamma}$. Fix an arbitrary processor v. By Theorem 5.7, for each $1 \le i \le |\theta|$, there exists an execution e_i with offset function δ_i, such that

$$\delta_i(p_{i-1}, p_i) = \overline{w}(p_{i-1}, p_i) \ . \tag{7.1}$$

Theorem 5.7 also says that the set of output values at v (being part of the basic state of the CSA at v), is the same in e and all the e_i. We therefore have that for each i,

$$\begin{aligned}
tightness_v(e) &= tightness_v(e_i) \\
&\ge now_{e_i}(p_{i-1}) - now_{e_i}(p_i) && \textit{correctness requirement} \\
&= \delta_i(p_{i-1}, p_i) + virt_del(p_{i-1}, p_i) && \textit{by definition of offset} \\
&= \overline{w}(p_{i-1}, p_i) + virt_del(p_{i-1}, p_i) && \textit{by Eq. (7.1)}
\end{aligned}$$

Summing the above over all i, we get

$$\begin{aligned}
|\theta| \cdot tightness(e) &\ge \sum_{i=1}^{|\theta|} \overline{w}(p_{i-1}, p_i) + \sum_{i=1}^{|\theta|} virt_del(p_{i-1}, p_i) \\
&= \sum_{i=1}^{|\theta|} \overline{w}(p_{i-1}, p_i) + \sum_{i=1}^{|\theta|} (local_time(p_{i-1}) - local_time(p_i)) \\
&= w(\theta) \ ,
\end{aligned}$$

because the second sum is cyclic. In other words, for any processor v, $tightness_v(e) \ge w(\theta)/|\theta|$. Since θ was an arbitrary cycle in $\overline{\Gamma}$, we conclude that $tightness_v(e) \ge \mathrm{mcm}(\overline{\Gamma})$, as desired. ∎

Theorem 7.1 coincides with known results for the special case of systems with drift-free clocks. For example, Lundelius and Lynch [19] considered a system of n processors, where the underlying communication graph is complete, and the latency bounds of all messages are finite and identical (say upper bound H and lower bound L). The corresponding synchronization graph consists of n points (one per processor), and between each pair of points p, q there are arcs (p, q) and (q, p) with weights satisfying $w(p, q) + w(q, p) = H - L$. It can be shown that for these graphs, the maximum cycle mean is $(H - L)(n - 1)/n$, which is the lower bound proved in [19].

Halpern, Megiddo and Munshi [13] extended the result of [19] to the case where the underlying graph of the system is not complete, and the latency bounds for each link may be different (i.e., there are different H and L for each link). Again, their lower bound can be viewed as showing that the worst possible scenario under the given constraints is bounded by the maximal cycle mean in the corresponding synchronization graph.

Attiya, Herzberg and Rajsbaum [3] refined the results of [13] to hold for each execution of the system, rather than for the worst possible executions. Theorem 7.1 generalizes the result of [3] to the case of bounded-drift clocks. Our result generalizes the previous bounds also to the case where the latency bounds may be different for each individual message.

Summary

In this chapter we discussed the internal clock synchronization problem. Formally, based on the definition of [13]. Using synchronization graphs, we presented a new lower bound for internal synchronization for system over systems with drifting clocks. This lower bound generalizes known lower bounds for systems with drift-free clocks to the general case of bounded-drift clocks.

Chapter 8

The Space Complexity of Optimal Synchronization

Call a synchronization algorithm *general* if it works for all possible environments as defined in Section 3.1, i.e., for all possible views, all possible message latency bounds, and all possible clock drift bounds. (For example, the full information protocol used in the proof of Theorem 6.4 is a general algorithm for external synchronization, whereas the algorithm described in Section 6.3 is not general, since it works only for drift-free clocks.) In this chapter we provide strong evidence that suggest that a general CSA for external synchronization which is optimal must be inefficient, or more specifically, such an algorithm cannot have bounded space complexity.

Recall that in external clock synchronization systems, the CSAs are required to compute bounds on the current reading of some designated drift-free clock called the *source clock* (see Section 4.1 for the full definition). In this chapter, we prove that for a certain reasonable computational model, there exist scenarios in which the space complexity required to compute optimal output cannot be bounded. The result is obtained in a small system (four processors, two of which have drift-free clocks).

The first problem in formalizing a space lower bound is that our model allows for real numbers: a real number can be used to encode an unbounded amount of information. Our strategy to get around this difficulty is to bound from below the number of "control bits" required to run the program, where we disallow fiddling with the input values.

The moral of the result presented in this chapter is that one cannot have a synchronization algorithm which is simultaneously optimal, general, and efficient. An algorithm designer must decide which of the three is to be sacrificed. We remark that as a by-product, this chapter indicates that the inefficiency of the algorithm used in the proof of Theorem 6.4 was, in a certain sense, unavoidable, since that algorithm is both general and optimal.

The remainder of the chapter is organized as follows. In Section 8.1 we describe the computational model in the context of CSAs, and in Section 8.2 we give the space lower bound proof.

8.1 The Computational Model

The model we use for computations of CSAs is a particular kind of the computation tree model. First, we define the following algebraic concept.

Definition 8.1 *A* special linear form *for a set $X = \{x_1, \ldots, x_N\}$ is a sequence of N integers $f = \langle c_1, \ldots, c_N \rangle$. The value of f under the assignment $x_1 = a_1, \ldots, x_N = a_N$ is $f(a_1, \ldots, a_N) = \sum_{i=1}^{N} c_i a_i$, where $a_i \in \mathbf{R} \cup \{-\infty, \infty\}$.*[1] *If $b = f(a_1, \ldots, a_N)$ for some special linear form f, then b is said to be a* special linear combination *of $a_1, \ldots, a_N$.*

We have the following simple lemma.

Lemma 8.1 *If b is a special linear combination of $a_1, \ldots, a_N$, and for each $i = 1, \ldots, N$ we have that a_i is a special linear combination of $a_{i1}, \ldots, a_{iK_i}$, then b is a special linear combination of $a_{11}, \ldots, a_{1K_1}, \ldots, a_{N1}, \ldots, a_{NK_N}$.*

Proof: Since $b = \sum_{i=1}^{N} c_i a_i$ for some integers c_i, and since for each i we have $a_i = \sum_{j=1}^{K_i} c_{ij} a_{ij}$, for some integers c_{ij}, we can write b as the special linear combination

$$b = c_1 c_{11} a_{11} + \cdots + c_1 c_{1K_1} a_{1K_1} + \cdots + c_N c_{N1} a_{N1} + \cdots + c_N c_{NK_N} a_{NK_N} . \quad \blacksquare$$

We now define the computational model. For simplicity of presentation, we present below a model for deterministic CSAs; the extension to non-deterministic CSAs is straightforward. A *program* for a CSA module is specified by a directed labeled tree, where the root of the tree is called the *start node*, and the edges are directed away from the start

[1] We use the conventions that for any finte number r, $r + \infty = \infty$, $r - \infty = -\infty$, $0 \cdot \infty = 0 \cdot (-\infty) = 0$, and $\infty - \infty$ is undefined.

node. Intuitively, nodes represent control configurations of the program, and executions of the program proceed by following a directed path in the tree, starting at the start node. Formally, let us call the nodes at even distance from the start node *even nodes*, and nodes at odd distance from the start node *odd nodes*. The subtree of depth two rooted at each odd node corresponds to an input action followed by an output action of the CSA, as dictated by the non-interfering filtering condition. Specifically, we define the node labels as follows (see Figure 8-1 for an example the first three layers of a program tree).

- Each odd node is labeled by an input action name and *input variables*, where the input variables contain the local time and bounds mapping values (specified later); we call these variables *local variables*. If the action is $Receive\_Aug\_Message(m, m')$, there are also *message variables*, which correspond to values in m'. We require that for each even node, there is exactly one child node for any possible input action.

- Each even node, except for the start node, is labeled by an output action name, a *computation predicate*, and some *output forms* according to the following rules.

  – The output action of an odd node corresponds to the input action of its parent in the tree according to the non-interfering filtering property, i.e., if the action of the parent is $Send\_Message(m)$, then all its children nodes have an action of the type $Send\_Aug\_Message(m, m')$, and if the action of the parent is $Receive\_Aug\_Message(m, m')$, then the action of all its children is $Receive\_Message(m)$.

  – For an even node p in the tree, let $X(p)$ denote the set of input variables in labels on the path from the start node to p. The computation predicate of p is an arbitrary predicate over $X(p)$, and the output forms associated with p are special linear forms for $X(p)$.

  For each even node q, for any possible assignment of values to $X(q)$, we require that there is exactly one computation predicate among its children that evaluates to TRUE.

An execution of the CSA in this model proceeds by moving a "token" (which represents the current control configuration) along the tree according the labels in the following way. Initially, the token is placed at the start node. Whenever an input action occurs, the token is moved down the tree to the odd node whose label matches the input action name. In addition, the input variables associated with the odd node are instantiated. Next, an even

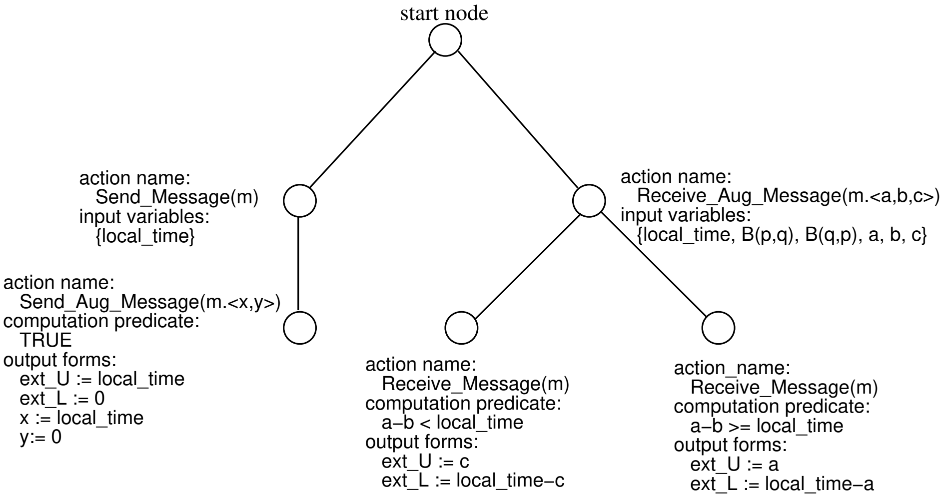

Figure 8-1: *The first three layers of a program: an example. The odd nodes are labeled by input action names and input variables, and the nodes at depth 2 are labeled by an output action name, a computation predicate and output forms.*

node down the tree is selected by choosing the node whose computation predicate evaluates to TRUE under the current assignment of the input values. The outcome of the predicates is well defined, as all their variables are instantiated at this stage. The output values are defined by instantiating the output forms associated with the chosen even node.

Let us now be more specific about the input variables and the output values of a program for a CSA. The input variables associated with an odd node, which in turn corresponds to an input step p, always include $local\_time(p)$, and the values of the standard bounds mapping of all the pairs (p, q) and (q, p), for all points q which are adjacent to p in the local view from p (if there are any). In addition, if p is a receive point, then the input also contains all the values that arrive in the incoming message. We restrict the message alphabet used by CSAs to be strings of $\mathbf{R} \cup \{-\infty, \infty\}$. The output forms associated with an even node which corresponds to a point p always contain forms for the mandatory output variables (i.e., ext_L and ext_U); if p happens to be a send point, then there is an output form corresponding to each value to be sent in the outgoing message. The output values of the CSA, at any state of the execution, are generated by instantiating the last output forms by the input values.

When time passage occurs, the local time and bounds mapping values are updated. Since these values may appear in the output forms for ext_L and ext_U, the output values are potentially updated as well. This completes the description of the way CSAs work in our model.

For lower bound purposes, we define the *space complexity* of a program in our model to be the logarithm to base 2 of the maximal degree of a node in the tree. We argue that this measure is certainly a lower bound on the number of bits required to distinguish among the different possible branches the program may take. We remark that in our proof, the lower bound is derived for the odd nodes, i.e., the number of possible output responses for an input.

Before we go into the lower bound proof, we state an important property of our model. First, we define the following concept.

Definition 8.2 *Let p be a point in a view $\mathcal{V}$ of an execution of a clock synchronization system. The* values in the local view of *p is the set of all local times of points in the local view* $\texttt{prune}(\mathcal{V}, p)$*, and all the bound mapping values for arcs* $\texttt{prune}(\mathcal{V}, p)$*.*

The important property of values in a local view of a point is that they "span" all possible outputs at that point, as stated in the following lemma.

Lemma 8.2 *Any output value of a CSA at a point p in an execution of the system is a special linear combination of the values in the local view of p.*

Proof: By induction on the points in the view, sorted by their order of occurrence in the execution. The lemma is clearly true in the first step of the execution in the system: the only input value at that point is the local time of occurrence, and by definitions, the output value is just a special linear combination of its input values.

Assume now that the lemma holds at all points $p_1, \ldots, p_n$ of the execution, and consider the point p_{n+1}. By Lemma 8.1, it is sufficient to show that the input values are special linear combination of values in the local view of p_{n+1}. If p_{n+1} is not the first action at the processor, let p_j be the previous action at the processor, and let p_j be undefined otherwise. We distinguish between two cases.

Case 1: p_{n+1} is a send point. In this case, by our model definitions, the input values at p_{n+1} are $\mathit{local\_time}(p_{n+1})$, and if p_j is defined, the input also contain the values of the standard bounds mapping for (p_{n+1}, p_j) and (p_j, p_{n+1}). Trivially, all these values are special linear combinations of values in the local view of p_{n+1}.

Case 2: p_{n+1} is a receive point. Let p_i denote the corresponding send point in the execution. The input values in this case are the local time of occurrence of p_{n+1}, the appropriate bounds mapping values, and the values that arrive in the incoming message. Since a send point always occurs before the corresponding receive point, we have that $i < n+1$, and by definition, we also have that the local view of p_i is contained in the local view of p_{n+1}. By the inductive hypothesis, the values that arrive in a message are special linear combinations of values in the local view of p_i, and hence they are also special linear combinations of values in the local view of p_{n+1}. This completes the inductive step. ▮

8.2 The Space Lower Bound

In this section we prove a lower bound on external synchronization in the model defined in previous sections. We shall use the following simple lemma.

Definition 8.3 *A function $F : D \mapsto R$ is said to be* covered *by a collection of functions $\mathcal{F}$ if for all $x \in D$ there exists a function $f \in \mathcal{F}$ such that $F(x) = f(x)$.*

Lemma 8.3 *Let $\overline{x}_1, \ldots, \overline{x}_M \in \mathbf{R}^N$ be such that for any $\overline{x}_i = (x_{i1}, \ldots, x_{iN})$ and $\overline{x}_j = (x_{j1}, \ldots, x_{jN})$ we have that if $x_{ik} \neq x_{jk}$ then $x_{ik} - x_{jk}$ is an integer. Let F be a function such that $F(\overline{x}_i) - F(\overline{x}_j)$ is an integer only if $i = j$. If $\mathcal{F}$ is a collection of special linear forms covering F, then $|\mathcal{F}| \geq M$.*

Proof: By contradiction. If $|\mathcal{F}| < M$ and $\mathcal{F}$ covers F, then for some $f \in \mathcal{F}$ and $i \neq j$, we have that $f(\overline{x}_i) = F(\overline{x}_i)$ and $f(\overline{x}_j) = F(\overline{x}_j)$. Denote $f = \langle c_1, \ldots, c_N \rangle$, $\overline{x}_i = (x_{i1}, \ldots, x_{iN})$ and $\overline{x}_j = (x_{j1}, \ldots, x_{jN})$. Suppose, w.l.o.g, that $x_{i1} - x_{j1}, \ldots, x_{iK} - x_{jK}$ are all integers, and that $x_{in} = x_{jn}$ for $n = K+1, \ldots, N$. Then

$$\begin{aligned}
F(\overline{x}_i) - F(\overline{x}_j) &= f(\overline{x}_i) - f(\overline{x}_j) \\
&= \sum_{n=1}^{N} c_{in} x_{in} - \sum_{n=1}^{N} c_{jn} x_{jn} \\
&= \sum_{n=1}^{N} c_{in}(x_{in} - x_{jn}) \\
&= \sum_{n=1}^{K} c_{in}(x_{in} - x_{jn}) \ ,
\end{aligned}$$

which is an integer, contradicting the assumption that $F(\overline{x}_i) - F(\overline{x}_j)$ is not an integer for $i \neq j$. ▮

We now turn to prove a lower bound on the space complexity of optimal CSAs in our computational model. To simplify presentation, we focus below on the output variable *ext_L*.

Consider an execution of an external synchronization system, and let Γ be the synchronization graph generated by the local view of the execution at some point p and the standard bounds mapping. From Theorem 6.4, we know that the optimal value for *ext_L* at point p is precisely $local\_time(p) - d(sp, p)$, where sp is the source point of Γ, and d is the distance function of Γ. The lower bound is proven by showing that unbounded space is required to compute $d(sp, p)$ for a point p in a certain scenario.

Specifically, we consider a system whose underlying graph is a line of four processors denoted s, u, v, w (see Figure 8-2 (a)). Processor s is the source processor; processors u and v have drifting clocks, and the clock at w is drift-free. We concentrate on the CSA at w. As mentioned above, the optimal value of *ext_L* at a point p of the execution is $local\_time(p) - d(sp, p)$. Since $local\_time(p)$ is an input variable at p, the task we consider reduces, at each point p, to the computation of $d(sp, p)$.

The following key lemma describes a scenario in which a single local view may have many different extensions, depending on the message that arrives next. The output for each possible extension must be different; the special properties of the input variables at the receive point are used later to prove the space lower bound.

Lemma 8.4 *For any integer $M > 0$ there exist M executions $e_1, \ldots, e_M$ with views $\mathcal{V}_1, \ldots, \mathcal{V}_M$ and synchronization graphs $\Gamma_1, \ldots, \Gamma_M$, respectively, and a receive point p that occurs at w, such that*

(1) p is common to all views.

(2) The local views of $\mathcal{V}_1, \ldots, \mathcal{V}_M$ at w are identical before p occurs.

(3) All values in the message that arrive at p are integers.

(4) For each $i = 1, \ldots, M$, the distance between sp and p in Γ_i is $1/(i+1)$.

Proof: We construct the views, and specify the weights of the arcs in corresponding synchronization graphs as we go. In our construction, all arc weights are non-negative, and hence there are no negative-weight cycles in all the synchronization graphs we define. Therefore, the proof is completed by observing that by Theorem 5.7, for each i there exists an

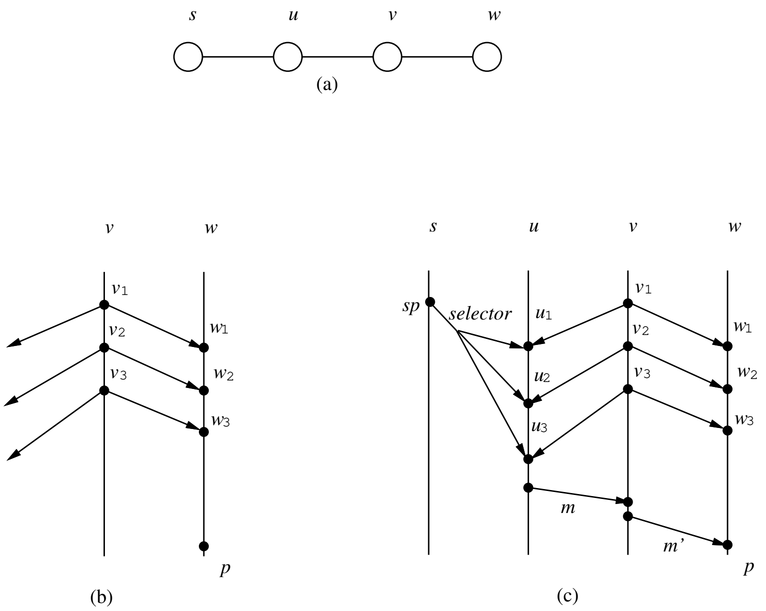

Figure 8-2: *(a) System structure for the proof of Lemma 8.4. Processor s is the source, and processor w also has a drift-free clock. (b,c) An example for graphs constructed in the proof of Lemma 8.4 with $M = 3$. In (b), the local view at w before p (shared by all $\mathcal{V}_i$) is illustrated (the messages from v are known to be sent). In (c), the local view at w after p is illustrated: in $\mathcal{V}_i$, the selector message is received at point u_i.*

execution e_i with view $\mathcal{V}_i$, such that e_i satisfies the bounds mapping derived from Γ_i and $\mathcal{V}_i$.

It remains to define the views and the bounds mapping. We do it as follows (see Figure 8-2 (c)). In all views $\mathcal{V}_i$ for $i = 1, \ldots, M$, there are M messages from processor v to processor u, with distinct send points denoted $v_1, \ldots, v_M$, and distinct receive points denoted $u_1, \ldots, u_M$, respectively. The bounds mapping is such that in all the Γ_i we have $w(v_k, u_k) = 0$, $w(u_k, v_k) = 1$ for $k = 1, \ldots M$, and $w(v_k, v_{k+1}) = w(v_{k+1}, v_k) = w(u_k, u_{k+1}) = w(u_{k+1}, u_k) = 1$ for $k = 1, \ldots, M-1$. Also, in all views $\mathcal{V}_i$ there are M messages sent from v to u with send points denoted $v_1, \ldots, v_M$, and receive points denoted $w_1, \ldots, w_M$, respectively. In all the Γ_i we have $w(w_k, v_k) = 1$ for all k. The weight of the arc (v_k, w_k) is defined to be $1/(k+1)$.

In addition, all views $\mathcal{V}_i$ have a message m sent from u to v after the last u_k point, and a message m' sent from v to w after m is received at v. The receive point of m' is the point p, promised in the statement of the lemma. The weight the four arcs corresponding to m and m' is 1 in all Γ_i.

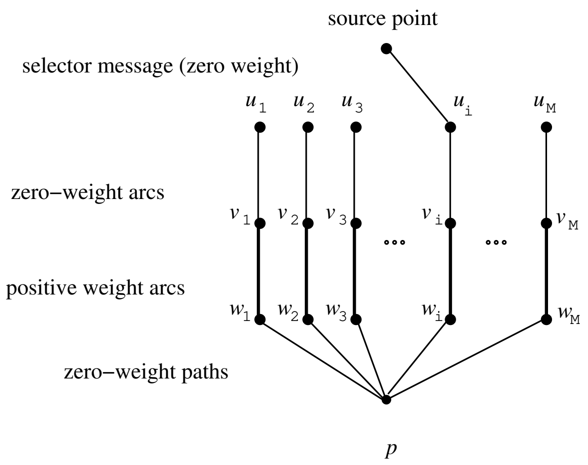

Figure 8-3: *A schematic summary of the distance situation for a typical view $\mathcal{V}_i$. The arcs that are not drawn have weight* 1*. The distance from the source point to p is $w(v_i, w_i) = 1/(i+1)$.*

Only the following feature differs in the different views $\mathcal{V}_i$: for each $i \in \{1, \ldots, M\}$, we have in view $\mathcal{V}_i$ a message, called the *selector message*, sent from the source processor at point sp and received at processor u at point u_i. In Γ_i, we have $w(sp, u_i) = 0$ and $w(u_i, sp) = 1$.

Finally, we choose the local times of all points in all views to be integers. Thus, the bounds mapping values, which are determined by the local times and the arc weights, are also all integers, except for the pairs (v_k, w_k) for $k = 1, \ldots, M$. This completes the description of the views $\mathcal{V}_i$.

We now observe that the views thus defined have the required properties. Parts (1) and (2) are immediate from the construction: p is common to all views, and the local view at w before p is identical for all $\mathcal{V}_i$ (see Figure 8-2 (b)). Part (3) of the lemma follows from Lemma 8.1 and the fact that by construction, all values in the local view at the point at which m' is sent are integers. Finally, Part (4) of the lemma is clear from the construction (see Figure 8-3). ▮

We can now prove the space lower bound.

Theorem 8.5 *Let A be a general external CSA. If A is an optimal algorithm (as defined in Def. 4.1), then its space complexity cannot be bounded by a function of the system size.*

Proof: Suppose A is a general optimal synchronization algorithm for external synchronization. Then by Theorems 6.3 and 6.4, at any point p that occurs at processor v in an execution, it must be the case that $ext\_L_v = local\_time(p) - d(sp, p)$, where d and sp are the distance function and the source point, respectively, in the corresponding synchronization graph. By Lemma 8.4, for any $M > 0$ there are M scenarios with a common point p such that at p, the local input variables are the same at all scenarios, the other input values are all integers, and such that in scenario i the optimal output is $local\_time(p) - 1/(i+1)$. Letting $\overline{x}_1, \ldots, \overline{x}_M$ denote the input values of these scenarios, and letting F denote the optimal value of ext_L, we can there apply Lemma 8.3, and deduce that there are at least M distinct output forms associated with p. It follows that the degree of the odd node in the program corresponding to p is arbitrarily large, and since the space complexity of a branching program is the logarithm of the maximal degree of a node, we conclude that the space required by the program cannot be bounded as a function of the network size. ∎

Remark. The crucial property of the model used in the lower-bound argument is the restriction that output is represented by special linear combinations. We argue that this restriction is reasonable for two reasons. First, we know that optimal output can be computed this way: synchronization distances can be expressed as special linear combinations of local times and bounds. And secondly, as already mentioned above, if we do not impose restrictions on the computational model, there is no hope for a space lower bound, since an unbounded amount of state information can be encoded in a single real number.

Summary

In this chapter we looked at the space complexity required to store the state of optimal CSAs for external synchronization. We defined a computational model, where output may be represented only by linear combination of the input values with integer coefficient. The program is represented by a tree, and the space complexity is the logarithm of the maximal branching factor in the tree. We then proved that there are executions of very simple systems (we used four processors), for which the space complexity of an optimal CSA cannot be bounded. This means that any optimal algorithm for external synchronization that works for all environments must have unbounded space complexity. The implication of this result is that there is no synchronization algorithm which is simultaneously efficient, optimal and general.

Chapter 9

Extensions

The analysis of synchronization graphs, presented in Chapter 5, was developed for the model of clock synchronization systems, as defined in Chapter 3. This model, while being arguably a reasonable abstraction of real systems, is restrictive. In this chapter we look at a few simple variants of the basic model, and show how using our concept of synchronization graph, one can analyze these variants quite easily.

Our discussion is presented in three parts. In Section 9.1 we consider the case of additional timing constraints. We show how a few kinds of additional timing constraints can be incorporated into synchronization graphs. In Section 9.2 we discuss timing faults, i.e., cases where an execution violates the system specification. We define a natural notion of detectable faults, and show that synchronization graphs can be used to detect the existence of such faults. In Section 9.3 we consider structured send modules, i.e., systems in which the message sending pattern has a more regular structure. Using a simple example, we explain how knowledge of the structure of the send modules can help in generating timing information without explicit communication.

9.1 Additional Timing Constraints

The definition of clock synchronization systems in Chapter 3 allows for two sources of timing information: the message latency bounds and the clock drift bounds. It is often the case that we have some additional sources of timing information. For example, the presence of a human operator at a site may suffice to insure that the absolute offset of the local clock at that site is never too big. Another example is a broadcast of a message to a subset of the processors, where it is known that the message is delivered at all processors within a period of known length (even though the time to deliver any individual message may be arbitrary). Having such additional information may improve the synchronization attained by CSAs. Below, we describe ways to incorporate a few simple types of such knowledge into synchronization graphs. By doing this, the distances in the synchronization graph have the additional information built into them, and can therefore be used to get better synchronization.

9.1.1 Absolute Time Constraints

Suppose we know somehow that "an event p occurs at real time at least a," or that "an event p occurs at real time at most b." Formally, we may have *absolute time constraints*, defined to be statements of the form

$$now(p) \in [a, b] \ ,$$

where p is a point in the view, and $[a, b]$ is a (possibly infinite) interval of real numbers.

Absolute time constraints can be incorporated in the synchronization graph as follows. We introduce a new point into the graph, called the *origin* and denoted by s_0, where for analysis purposes we assume that $local\_time(s_0) = now(s_0) = 0$. (Intuitively, the origin can be thought of as representing the initialization event of the execution.) For each absolute time constraint $now(p) \in [a, b]$, we introduce two arcs (p, s_0) and (s_0, p) into the synchronization graph, with weights

$$w(s_0, p) = -a \ , \quad \text{and} \quad w(p, s_0) = b \ .$$

It is easy to see, using Lemma 5.2 and the attributes of the origin as defined above, that the new arcs and weights express the given constraint. Bounds on relative offsets of the points in the view can now be obtained as usual, by finding distances between the desired points in the extended synchronization graph. In addition, bounds on the *absolute offsets* can be obtained by computing the distances to and from the origin: with the real and local time attributes we assigned to the source point, we have that for any point p, $\delta(p) = \delta(p, s_0)$, and hence $\delta(p) \in [-d(s_0, p), d(p, s_0)]$.

By adding the origin node and its incident edges, the distances in the synchronization graph may drop, resulting in tighter bounds on the offset between points, i.e., better synchronization.

9.1.2 Relative Time Constraints

Suppose that we have information of the type "at least a time units elapse between the occurrence of an event p until the occurrence of an event q," or "at most b time units elapse between the occurrence of an event p until the occurrence of an event q." Formally, we may have a *pairwise time constraint*, given as a statement of the form

$$now(q) - now(p) \in [a, b] \ .$$

Modeling pairwise time constraints is done using the tools we already have: the interpretation of such a statement is simply that the bounds mapping B of the pattern in question should be extended to include $B(q,p) = b$ and $B(p,q) = -a$. To translate this information into the distance measure of synchronization graphs, we augment the graph with arcs (p,q) and (q,p), and assign their weights as usual (see Def. 5.4). As before, the introduction of additional arcs into the synchronization graph may reduce the distances between points, thus resulting in tighter bounds on synchronization.

Another instance of relative time constraints is where a set of events is known to occur within a time interval of known length. (Halpern and Suzuki [12] make this assumption for the set of receive events of a broadcast message.) Formally, we have a set Q of events, such that for any pair $p_i, p_j \in Q$ we know that

$$now(p_i) - now(p_j) \leq a \ ,$$

and the reduction to pairwise time constraints is obvious.

Remark. It may be interesting to push further the idea underlying the simple technique suggested above for pairwise time constraints. The way we developed our model in Chapter 3, we had the natural notion of adjacent points (cf. Def. 3.9), and bounds mapping was defined only for pairs of adjacent points. This definition was motivated by the assumption that the only source for timing information are the specifications of local clocks and network links. The idea in the generalization suggested above is that the basic relation is pairwise time constraints, rather than adjacency. Put in other words, instead of defining bounds mapping in terms of the classical adjacency relation, we should define the adjacency relation in the synchronization graph in terms of the pairwise time constraints.

9.2 Fault Detection

Throughout the discussion of synchronization graphs we relied heavily on its "integrity," namely the fact that $act\_del(p,q) \leq B(p,q)$ for all adjacent points p, q. Since this assumption may not always hold — e.g., if some component of the system fails, or if the specification is simply wrong — it is interesting to understand what happens in that case. Fortunately, Theorem 5.4 guarantees a strong fault-detection property. Let us first define the a notion of *detectable fault.*

Definition 9.1 *Let $\mathcal{V}$ be a view and let B be a bounds mapping for $\mathcal{V}$. $\mathcal{V}$ is said to have a* detectable fault *with respect to B if there is no pattern with view $\mathcal{V}$ that satisfies B.*

Using Theorem 5.5, we derive the following result.

Lemma 9.1 *Let $\mathcal{V}$ be a view of an execution of a clock synchronization system, and let B be a bounds mapping for $\mathcal{V}$. Then $\mathcal{V}$ has a detectable fault with respect to B if and only if the synchronization graph Γ defined by $\mathcal{V}$ and B contains a negative weight cycle.*

Proof: Suppose first that Γ contains a negative cycle. Then it follows from Theorem 5.6 that there is no pattern with view $\mathcal{V}$ that satisfies B, and hence $\mathcal{V}$ has a detectable fault w.r.t. B. Conversely, suppose that Γ does not contain a negative-weight cycle. If Γ is empty, then trivially $\mathcal{V}$ does not contain a detectable fault w.r.t. B, and we are done. Otherwise, let p_0 be any point in Γ. By Theorem 5.7, there exists at least one pattern $\mathcal{P}$ with view $\mathcal{V}$ such that $\mathcal{P}$ satisfies B, and hence $\mathcal{V}$ has no detectable faults w.r.t. B. ▮

We remark that algorithms that use our techniques, probably compute distances over the synchronization graph anyway. Since shortest paths algorithm for general edge weights usually discover negative weight cycles, we get fault detection "for free." However, we remark that we do not know of a general technique for *fault correction* using synchronization graphs directly.

9.3 Structured Environments

The basic theory studies the case where send modules are completely unstructured (technically, the "send" action is always enabled), and where the link automata may lose messages arbitrarily. Somewhat surprisingly, it turns out that one may gain timing knowledge also from the *absence* of a message receive event, in the case of reliable communication.[1]

We now explain how can one add arcs to the synchronization graph for messages which are guaranteed to arrive, but haven't arrived. Again, the extra arcs may result in shorter distances and hence better synchronization.

In the following lemma, we assume that the drift upper bound of one of the clocks is at least 1. This can be done without loss of generality since local time readings can be scaled to satisfy this assumption.

Lemma 9.2 *Suppose that the send module at processor u is such that a message m is always sent at a point q with known local time, suppose that the link automaton L_{uv} is such that m is guaranteed to be always received at processor v within $H(m)$ time units, and suppose further that the drift upper bound of the clock at v satisfies $\overline{\varrho}_v \geq 1$. Then for any point p at v where m has not yet been received we have $\delta(p,q) \leq H(m) - virt\_del(p,q)$.*

Proof: Consider the point p' in which m is received at v. By assumption, $\overline{\varrho}_v \geq 1$. Since p occurs at v before p', we have $local\_time(p') \geq local\_time(p)$, and hence $virt\_del(p',q) \geq virt\_del(p,q)$ and $virt\_del(p',p) \geq 0$. Therefore, using Def. 3.11 and Lemmas 5.1 and 5.2, we get

$$\begin{aligned}
\delta(p,q) &= \delta(p,p') + \delta(p',q) \\
&\leq (1 - 1/\overline{\varrho}) \cdot virt_del(p',p) + H(m) - virt_del(p',q) \\
&\leq H(m) - virt_del(p,q)\ .
\end{aligned}$$

∎

The consequence of Lemma 9.2 is that if communication links do not lose messages and have finite latency upper bounds, one can add points and arcs to the synchronization graph,

[1]The place where the fact that messages may be arbitrarily lost was used in the proof of Theorem 3.2, where we proved that any local view at a point is also a complete view of some execution. This theorem does not hold in the case where some messages are guaranteed to be delivered: a local view that contains only the send point of such a message is not the complete view of any execution.

even if these points are not in the local view. Using the notation of Lemma 9.2, although q is not a part of the local view at p, the synchronization graph at p might as well include q and an arc (q, p) whose weight is $w(q, p) = H(m) - virt\_del(p, q)$ (since we have a pairwise time constraint between p and q).

Summary

In this chapter we discussed a few simple extensions of the basic model. We showed how to incorporate additional assumptions, such as absolute time constraints an relative time constraints into the synchronization graph. Such constraints may be known due to unmodeled parts of the system.

We also proved a strong fault detection capability for synchronization graphs. Despite the fact that we do not know how to exploit a synchronization graph directly for error correction, we get fault detection essentially for free.

Finally, we showed that if the send module is structured in a certain simple sense, and if communication links are reliable, then some timing information may be derived even from absence of messages. We showed how to incorporate such information into the synchronization graph.

These examples demonstrate the robustness of the basic concept of synchronization graphs. Many more variants are possible (e.g., finite granularity clocks, and external synchronization systems with multiple sources).

Chapter 10

Conclusion

Our hope is that the main contribution of this thesis is improved understanding of the clock synchronization problem. We believe that the insight developed in this thesis may lead to better synchronization protocols. We have suggested a new viewpoint for the problem, and presented new analytical tools and algorithmic techniques to deal with clock synchronization. Our results indicate that there is no "ultimate solution" for clock synchronization, but they leave hope that optimal efficient algorithms can be found for particular systems, or that better algorithms can be developed for general systems. For example, it seems reasonable to assume that our techniques can be implemented over the Internet, thus improving on the current version of NTP [26]. In addition, by implementing our methods with bounded space, one can get algorithms which are optimal with respect to a part of the execution (e.g., an algorithm that guarantees that its output is the best possible output for the last day).

On the theoretical side, we believe that synchronization graphs may prove a useful tool in the analysis of timing-based systems. In a sense, synchronization graphs can be viewed as a weighted version of Lamport's graphs [16]: Lamport used his unweighted graphs to describe executions of completely asynchronous systems; synchronization graphs are weighted, and can be used to describe executions of systems where processors have clocks.

Let us review the main weaknesses of synchronization graphs. Informally, the usefulness of synchronization graphs relies on a few strong assumptions.

(1) The system specification is such that if an event may occur at either of two points, then this event may occur at any time between them.

(2) Processors follow the system specification.

(3) All executions that satisfy the system specifications are possible.

As we mentioned in this thesis, assumption (1) cannot be compromised by our analysis. Without it, clock synchronization problems cannot even be expresses as linear programs. Regarding assumption (2), we gave a partial answer for the problem of systems that do not adhere to their specification by showing that synchronization graphs can be used for fault detection. We hope the error correction can also be aided by synchronization graphs. Assumption (3) leaves room for specializing the synchronization graphs according to the particular system being considered. We demonstrated such adaptations with a few simple examples.

Since clock synchronization is used throughout the spectrum of distributed systems — starting from a single VLSI chip, and ranging up to a global network — it is conceivable that the effect of even a slight improvement in the tightness of synchronization may be sweeping. For example, tighter synchronization of the transmitting and receiving endpoints of communication links can lead to better utilization and hence larger throughput of the communication network; better synchronization may imply shorter processing time for large databases. We hope that despite its weaknesses, this thesis can be used to improve synchronization in many cases. This may lead to a slightly more convenient world, and it can perhaps be translated into financial profit (for example, Merrill Lynch is using NTP to synchronize their worldwide network [11]).

It may be interesting to note that after our preliminary paper [29] was published, a few papers which have considerable overlap with our results have appeared. Specifically, Dolev *et al.* [8] have defined the notion of observable clock synchronization which is closely related to our notion of optimal clock synchronization. Their analysis is for the special case where the communication is done over a broadcast channel. Moses and Bloom [27] look at the problem of clock synchronization from the knowledge theoretic perspective. They study the case of drift-free clocks, and their main result can be viewed as a special case of one of our characterization theorems. Ajtai *et al.* [2] present an approach for the analysis of distributed algorithms which is closely related to our notion of local competitiveness.

Let us conclude with some interesting problems that this thesis leaves unsolved.

Fault Resilience: It would be interesting to develop a technique that uses synchronization graphs in the presence of errors, such that erroneous data can be overcome, more than merely detecting the existence of an error.

Internal synchronization: We do not know of a good technique for on-line distributed internal synchronization other than the naive use of external synchronization algorithms. Conceivably, synchronization graphs can be used to this end.

Appendix A

Time-Space Diagrams

In this appendix we present *Time-Space Diagrams* [17]. This representation method is a convenient way to graphically draw and view executions of distributed systems. (See Figure A-1 for an example.) The idea is that the x coordinate is used to denote location in space (which is, in the context of distributed systems, simply a *processor name*), and the y coordinate is used to denote real time. Since the physical location of processors is immaterial, processors are represented by vertical lines labeled by their names. In our diagrams we follow the convention that time grows downwards.

Given an execution of a system, its time-space diagram is drawn by the following two rules. First, the events of the execution (such as message send and receive) are represented by points, and hence the (x, y) coordinates of each event are determined by its location and time of occurrence. And secondly, a message is represented by a directed arrow, that

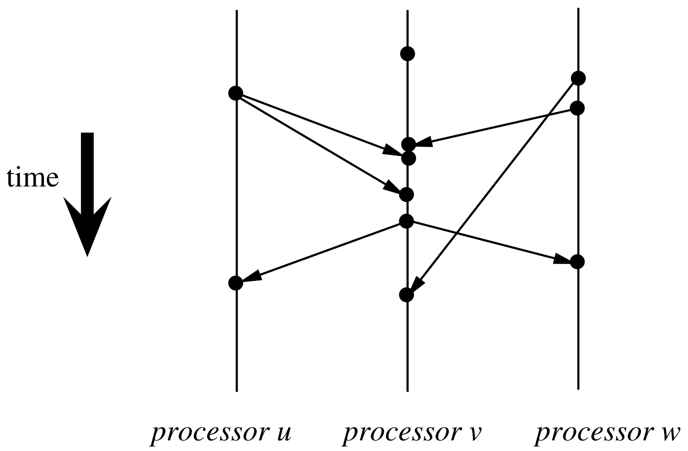

Figure A-1: *An example of a time-space diagram.*

connects the point corresponding to its send event to a point corresponding to its receive event. We can model in this way many types of communication assumptions, including broadcast (for example, in Figure A-1 processor v sends messages simultaneously to u and w), message duplication (in Figure A-1 there are two receive events at v that correspond to a single send event at u), message re-ordering (the messages sent by w in Figure A-1 are received in reversed order at v), and message loss (the first event at v in Figure A-1 might be a send event of a message which is not received).

Bibliography

[1] J. E. Abate, E. W. Butterline, R. A. Carley, P. Greendyk, A. M. Montenegro, C. D. Near, S. H. Richman, and G. P. Zampelli. AT&T's new approach to the synchronization of telecommunication networks. *IEEE Communication Magazine*, pages 35–45, Apr. 1989.

[2] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico*, Oct. 1994. To appear.

[3] H. Attiya, A. Herzberg, and S. Rajsbaum. Optimal clock synchronization under different delay assumptions. *SIAM J. Comput.*, 1994. Accepted for publication. A preliminary version appeared in *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, 1993.

[4] D. Bertsekas and R. Gallager. *Data Networks.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1992.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[6] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[7] D. Dolev, J. Y. Halpern, and R. Strong. On the possiblity and impossibility of achieving clock synchronization. *J. Comp. and Syst. Sci.*, 32(2):230–250, 1986.

[8] D. Dolev, R. Reischuk, and R. Strong. Observable clock synchronization. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994.

[9] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 70–75, 1982.

[10] R. Gawlick, R. Segala, J. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT Lab. for Computer Science, Dec. 1993.

[11] J. D. Guyton and M. F. Schwartz. Experiences with a survey tool for dicovering Network Time Protocol servers. Techical Report CU-CS-704-94, University of Colorado, Boulder, Jan. 1994.

[12] J. Halpern and I. Suzuki. Clock synchronization and the power of broadcasting. In *Proc. of Allerton Conference*, pages 588–597, 1990.

[13] J. Y. Halpern, N. Megiddo, and A. A. Munshi. Optimal precision in the presence of uncertainty. *Journal of Complexity*, 1:170–196, 1985.

[14] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[15] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Trans. Comm.*, 36(8):933–939, Aug. 1987.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.

[17] L. Lamport. The mutual exclusion problem. Part I: A theory of interprocess communication. *J. ACM*, 33(2):313–326, 1986.

[18] B. Liskov. Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, 6:211–219, 1993. Invited talk at the 9th Annual ACM Symposium on Principles of Distributed Computing, 1990.

[19] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Computation*, 62(2-3):190–204, 1984.

[20] N. Lynch. Simulation techniques for proving properties of real-time systems. In *Rex Workshop '93*, Lecture Notes in Computer Science, Mook, the Netherlands, 1994. Springer-Verlag. To appear.

[21] N. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1992.

[22] N. Lynch and F. Vaandrager. Forward and backward simulations – Part I: Untimed systems. Submitted for publication. Also, MIT/LCS/TM-486.

[23] M. Manasse, L. McGeoch, and D. Sleator. Competitive algorithms or on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing.* ACM SIGACT, ACM, May 1988.

[24] K. Marzullo and S. Owicki. Maintaining the time in a distributed system. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 44–54, 1983.

[25] D. L. Mills. Internet time synchronization: the Network Time Protocol. *IEEE Trans. Comm.*, 39(10):1482–1493, Oct. 1991.

[26] D. L. Mills. The Network Time Protocol (version 3): Specification, implementation and analysis. RFC 1305 RFC 1305, Network Working Group, University of Delaware, Mar. 1992.

[27] Y. Moses and B. Bloom. Knowledge, timed precedence and clocks. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994.

[28] Y. Ofek. Generating a fault tolerant global clock using high-speed control signals for the MetaNet architecture. *IEEE Trans. Comm.*, Dec. 1993.

[29] B. Patt-Shamir and S. Rajsbaum. A theory of clock synchronization. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montreal , Canada*, pages 810–819, May 1994.

[30] F. B. Schneider. Understanding protocols for Byzantine clock synchronization. Research Report 87-859, Department of Computer Science, Cornell University, Aug. 1987.

[31] B. Simons, J. L. Welch, and N. Lynch. An overview of clock synchronization. Research Report RC 6505 (63306), IBM, 1988.

[32] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.

[33] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

Index