Optimal and Efficient Clock Synchronization Under Drifting Clocks

Extended Abstract

Rafail Ostrovsky[*] Boaz Patt-Shamir[†]

February 24, 1999

Abstract

We consider the classical problem of clock synchronization in distributed systems. Previously, this problem was solved optimally and efficiently only in the case when all individual clocks are non-drifting, i.e., only for systems where all clocks advance at the rate of real time. In this paper, we present a new algorithm for systems with drifting clocks, which is the first optimal algorithm to solve the problem efficiently: clock drift bounds and message latency bounds may be arbitrary; the computational complexity depends on the communication pattern of the system in a way which is bounded by a polynomial in the network size for most systems. More specifically, the complexity is polynomial in the maximal number of messages known to be sent but not received, the relative system speed, and time-stamp size.

Our result has two consequences. From the theoretical standpoint, it refines the known bounds for optimal synchronization. But even more importantly, it enables us to derive new optimal algorithms that are reasonably efficient for most practical systems.

1 Introduction

Suppose that all processors in the system are equipped with clocks with known upper and lower bounds on their rate of progress. Suppose further that for each message delivered by the system, we are given lower and upper bounds on the transit time of the message. Suppose that one of the clocks, called *source*, is progressing precisely at the rate of real time (in fact, we may assume that the source determines what real time is), and the task of all other processors

[*]Bellcore Morristown, NJ E-mail: rafail@research.bellcore.com

[†]Research partly done while the second author was visiting Bellcore. Partially supported by a grant from Israel Ministry of Science and Technology. Dept. of Electrical Engineering–Systems Tel-Aviv University E-mail: boaz@eng.tau.ac.il

is to provide the users with the best estimate of the source clock, where an estimate is an interval which is guaranteed to contain the source time. This task is usually referred to as *external synchronization.* External clock synchronization is solved by a *clock synchronization algorithm* (abbreviated *csa* henceforth) by means of clock readings, message exchange, and system specification, including the bounds on clock rates and message speeds. Now, a csa is said to be *general* if it applies to all systems, where the real-time specifications are given as unrestricted non-negative parameters (including infinity). A csa is said to be *optimal* if at all times, any tighter estimate (i.e., a smaller interval) may be wrong on some execution which looks indistinguishable at that point. In [20, 19] the following basic facts were proven. First, there exists a general optimal on-line algorithm for external clock synchronization; furthermore, this existential result is complemented with a lower bound which shows that the worst-case space complexity (in a conservative computational model) of an optimal algorithm for general systems cannot be less than the logarithm of the number of events in the execution of the system. Indeed, the complexity of general optimal algorithm in [20] grew without bound as the number of events in the execution grew.

However, real clock synchronization algorithms need not be *general*: they should work for the particular system in which they are deployed. In many cases, such systems permit some significant assumptions on the timing specifications, which allow us to design efficient optimal algorithms. An important special case considered by many is systems with *drift-free* clocks, i.e., clocks which advance at the rate of real time (this assumption can be viewed as an abstraction of systems with short duration, where the drift of the clocks does not accumulate to more than a negligible amount). In [20] an efficient algorithm is presented for this case, based on the Bellman-Ford algorithm, where edge weights are derived from the local clocks readings and the real-time specifications of the system. It is not difficult to adapt this simple algorithm to scenarios where clocks drift by running a new version of the algorithm every short while (say, every hour), and combining the results by adding a "fudge factor" to account for the drift. Such implementations may beat other practical algorithms, but they are still not optimal [18].

In this paper we prove, by presenting a new algorithm, that clocks in many systems can be synchronized optimally *and* reasonably efficiently. The key to the new algorithm is a reduction of the synchronization problem to a single-source shortest-paths dynamic graph problem, which can be garbage-collected by using an all-pairs shortest-paths algorithm. To explain the complexity of the algorithm, we use Lamport's "happened before" relation [11]: the complexity of the algorithm at a node v is polynomial in the maximum, over all times t, of the number of "live messages" from the view point of v at time t: these are messages whose sending "happened before" t at v and whose delivery did not "happen before" t at v. In many practical systems, this number is linear in the size of the system. The complexity also depends polynomially on the relative system speed: this is the maximal number of events occurring at the system between any two consecutive events occurring at the same processor. Typically, the relative system speed is linear in the number of processors in the system. Note that relative system speed is related to (but different than) drift.

Related work. The work most relevant to the results presented in this paper is by Patt-Shamir and Rajsbaum [20, 19], which is summarized in Section 2.3 below. A closely related paper is by Moses and Bloom [17], where the upper bound on external synchronization is derived using a calculus of time bounds. Dolev et al. [8] use similar techniques to obtain synchronization algorithms for relativistic systems. Much effort has been devoted to studying *internal synchronization*, where the goal is to synchronize clocks within a system in which real-time is not available (see, e.g., [11, 12, 6, 10, 24, 1], surveys [22, 21] and references therein). The approach of comparing the synchronization bounds to the best possible bound for the given execution was first presented by Attiya et al. in [1], where they studied internal synchronization. The work in [1] extended the work of Halpern et al. [10], which analyzed internal synchronization as a "game against nature," which means that it is assumed that the execution should be taken as if it is generated by an adversary whose aim is to provide as little information as possible within the system specification. The work in [10], in turn, extended the work of Lundelius and Lynch [12], which analyzed internal synchronization in fully connected systems where all link specifications are identical. Much work is also devoted to the issue of fault-tolerant clock synchronization (e.g., [21, 13, 7]), which falls outside the scope of this paper.

As for practical work, two prominent approaches for clock synchronization are the NTP [15, 16] by Mills, used over the Internet, and probabilistic clock synchronization algorithm by Cristian [5]. We explain them briefly in Section 4.

Paper organization. In Section 2 we outline the basic system assumptions we use and review a few results directly relevant to our work. In Section 3 we present our main result, the new synchronization algorithm. In Section 4 we show that the new algorithm, under some properties shared by most systems, has polynomial complexity.

2 Model and Preliminaries

Our system model is the usual network model used, e.g., in [1, 20, 19]. In this section we give a summary of the essential assumptions. The system consists of *processors* and *communication links*. Processors are assumed to have unique identifiers. Communication links carry *messages* between processors. We assume that communication is reliable, and that links are bidirectional. Message receives and sends are called *events* or *points*, and we shall denote them by the letters p, q, r etc. Processors will be denoted by the letters u, v, w etc. For each event p there is a unique processor in which it occurs, denoted $\text{loc}(p)$. An *execution* of the system is a sequence of events, where each event p has its *real time of occurrence*, denoted $RT(p)$. (The real time attribute is used only for analysis, and is not available to processors: see "view" below.) In addition, we shall have for each event p and processor v the *local time of v at the occurrence of p*, denoted $LT_v(p)$; if p occurs at v, then we sometimes write $LT(p)$ for $LT_v(p)$. We shall think of an execution as a graph, whose node set is the set of events, with an edge (p, q) if either (i) q

is a receive event of the message whose send event is p, or (ii) p, q occur at the same processor and q is the first event following p. We further define the standard *happens before* relation between two events p, q, denoted $p \to q$, to hold if and only if there exists a (possibly empty) directed path from p to q in the execution graph. (Such graphs are sometimes referred to as the *Lamport graphs*, as they, as well as the happens before relation, were first defined by Lamport [11].) Note that in our execution graphs, nodes are also labeled by the real and local times of occurrence. A *view* of an execution is just an execution where the real time attributes are projected away. Thus a view can also be represented as a graph. Two executions are said to be *indistinguishable* if they share the same view. The intention is to capture the notion that real time is not available from within the system: a view of an execution contains only attributes available to the processors. To capture the notion that a processor has only local information regarding the execution, we define the notion of a *view from a point* p of an execution to be the view induced by points q such that $q \to p$.

We shall associate with each view a set of *real-time specifications*, which is a set of bounds on the difference of real times between pairs of events. We will mainly consider the following types of real-time specifications.

- Message transit bounds. Denote the send event of a message by p and its receive event by q. In any physical system, we have that $RT(q) - RT(p) \in [0, \top]$, but in many systems, non-zero lower bounds and finite upper bounds may be known.
- Clock drift bounds. Processor clocks usually have known bounds on their rate of progress with respect to real time. A typical workstation may have a quartz clock whose accuracy is 50 parts per million (abbreviated *ppm*), which means, for example, that if it shows that 10^6 time units have passed between events p and q, then we are guaranteed that $RT(p) - RT(q) \in [999950, 1000050]$.

Given a view, we represent the real time specifications uniformly by a *bounds mapping*, which is a function from pairs of events to $BR \cup \{\top\}$. The interpretation of a bounds mapping is just upper bounds: We shall say that an execution α *satisfies* a bounds mapping B if for all events p, q in the execution we have that $RT(p) - RT(q) \leq B(p, q)$. Note that lower bounds are also implicitly represented by bounds mappings: if we know that $RT(p) - RT(q) \in [L, U]$, then this fact is equivalent to having $B(p, q) = U$ and $B(q, p) = -L$. Under our usual real-time specifications, the bounds mapping value for a pair of points may be finite only if they are connected by an edge in the view graph.

Example. Consider a system with processors whose clocks may drift up to 100 ppm, and where message delivery time is completely arbitrary. Then an execution of this system can be modeled by the following bounds mapping B. $B(p, q) = 0$ if p is a send event and q is the receive event of that message; if p occurs after q at the same processor, then $B(p, q) = 1.0001 \cdot (LT(p) - LT(q))$ and $B(q, p) = 0.9999 \cdot (LT(q) - LT(p))$; for all other cases, $B(p, q) = \top$.

We stress that in our model, real-time specifications express only upper and lower bounds on the real times between pairs of events. We do not treat cases such as $RT(p) - RT(q) \in [L_1, U_1] \cup [L_2, U_2]$ with $L_2 > U_1$. (Such restrictions may arise if local clocks are not continuous.)

To show optimality of synchronization algorithm, we need an additional assumption which in effect says that the real time specifications expressed by the bounds mapping are the only criterion to determine whether an execution is possible. Formally, we make the following assumption. Let β be a view, and let B be a bounds mapping for β. If there is a way to assign real times to all points in β without violating B, then the result is a possible execution of the system.

2.1 External Synchronization

In an external synchronization system, one of the processors is designated as the *source* processor, and its clock is assumed to run at the rate of real time. The goal of all other processors is to get, at all points, the tightest estimate of the value source clock at that point. Formally, each processor is required to maintain two variables ext_L, ext_U such that at all times, the source clock is in the interval $[ext\_L, ext\_U]$.

External synchronization models systems where at least one of the processors has access to standard time as produced by a radio clock or an atomic clock. The concept is useful in loosely coupled systems such as the Internet: NTP is an external synchronization system [15, 16].

2.2 Synchronization Algorithms

In this work we consider only how to interpret the data collected by synchronization algorithms. In order to model this problem, we define passive clock synchronization algorithms, that do not affect system execution. In other words, the part we call "synchronization algorithm" does not initiate message sending, nor can it influence the real-time specifications of the system. This approach (proposed first by [1]) has the advantage of facilitating comparison of different algorithms under the same conditions: otherwise, if we allow an algorithm to create its own message traffic, it is not clear how to compare its results with results produced by another algorithm which generates a different type of message traffic. Furthermore, we restrict our attention to on-line distributed algorithms (in line with [20], and in contrast to [1]). This is modeled as follows (see Figure 1). Messages are generated and absorbed by a module called the *send module*, which abstracts the module which generates messages for the clock synchronization algorithm. Messages are delivered by the *network module*. The *clock synchronization algorithm* (abbreviated *CSA*) is a layer between the send module and the network. The CSA fills information in outgoing messages, and reads the information from incoming messages.

Thus, the only input a CSA module can have at its disposal at any point p is a complete local view of the execution from p, and the bounds mapping associated with that view (as it is

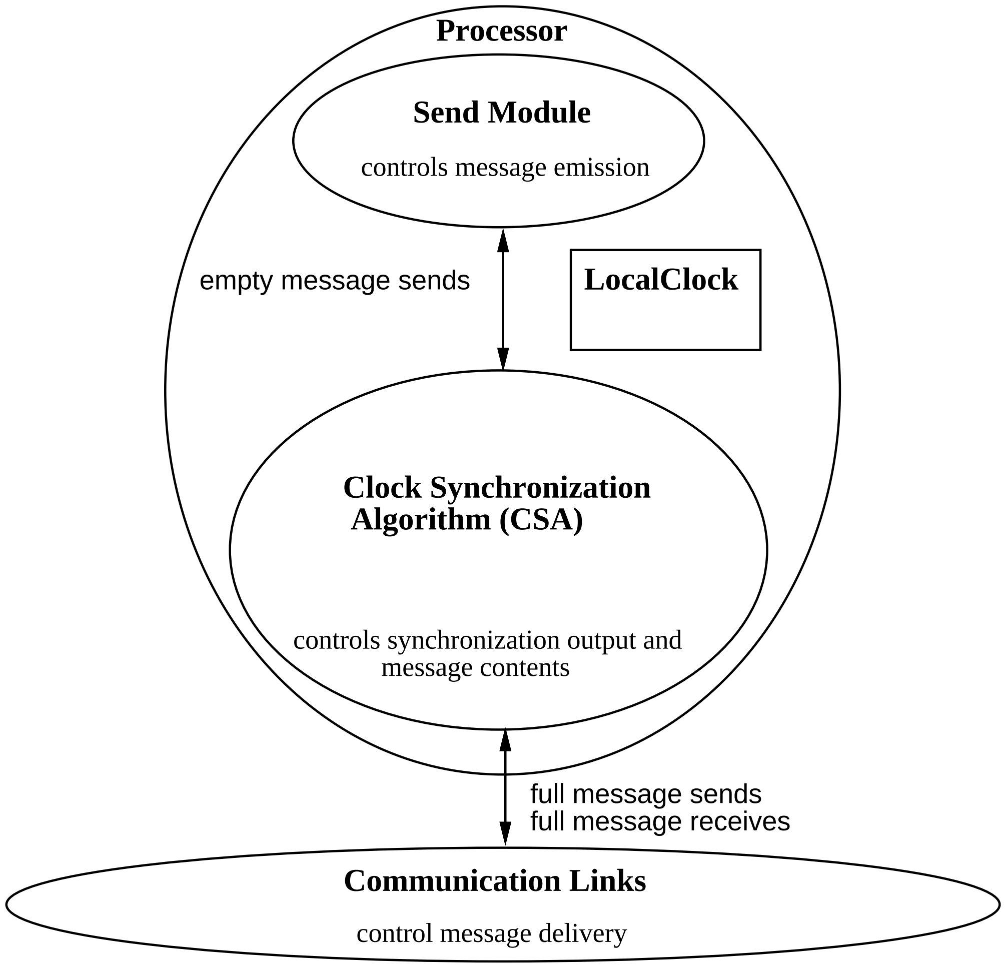

Figure 1: Schematic arrangement of the modules of a processor in a clock synchronization system.

derivable from the system specifications).

An external synchronization algorithm is called *general* if it works for any bounds mapping, i.e., it does not rely on assumption other than those listed above. An algorithm is called *optimal* if at all points, the difference $ext\_U - ext\_L$ is minimal in the sense that for any output $[a, b]$ with $b-a < ext\_U - ext\_L$, there exists an execution which is indistinguishable from that point, and such that the source time is outside the interval $[a, b]$.

2.3 Overview of Known Results

For the purpose of the unfolding discussion, it is convenient to use the notion of *virtual delay* between points, defined as the difference in their local times. Formally, $virt\_del(p, q) = LT(p) - LT(q)$. The concept of *synchronization graph* [20] is defined as follows.

Definition 2.1 *Given a view β and bounds mapping B for β, the* synchronization graph *for β and B is a weighted directed graph, denoted $\Gamma_{\beta B} = (V, E, w)$, where V is the set of events in the execution, $(p, q) \in E$ if and only if $B(p, q) < \top$, and $w(p, q) = B(p, q) - virt\_del(p, q)$.*

The synchronization graph translates the synchronization problem into a distance computation problem, as implied by the following theorem [20].

Theorem 2.1 (Clock Synchronization Theorem) *Let β be a view, B a bounds mapping for β, and let p, q be two points in β. Let d be the distance function in the synchronization graph $\Gamma_{\beta B}$. Then for all executions α with view β satisfying B we have that*

$$RT(p) - RT(q) \in [virt\_del(p, q) - d(q, p), virt\_del(p, q) + d(p, q)] \ .$$

Furthermore, there exist executions α_0 and α_1 with view β satisfying B such that in α_0 we have $RT(p) - RT(q) = virt\_del(p, q) - d(q, p)$, and in α_1 we have that $RT(p) - RT(q) = virt\_del(p, q) + d(p, q)$.

The clock synchronization theorem essentially says that the output of an optimal clock synchronization algorithm is simply distances in the corresponding synchronization graph. These distances are easily computable in principle, as demonstrated by the following algorithm.

General optimal algorithm for external synchronization. Send, in every message, the complete local view from the send point. Merge local views in the natural way. At any point, compute the synchronization graph defined by the local view from that point and the associated bounds mapping. Set $ext\_L = LT(p) - d(sp, p)$ and $ext\_U = LT(p) + d(p, sp)$, where sp is any point which occurs at the source processor, and d is the distance function in the synchronization graph.

It is easy to verify (by substituting $LT(sp) = RT(sp)$) that the bounds in the algorithm above coincide with the bounds in Theorem 2.1, and hence it attains optimal synchronization bounds. However, the algorithm is hardly practical: the size of the synchronization graph is usually linear in the number of the events in the execution (this is the case if the real-time specifications are given for messages and bounded-drift clocks). The complexity of the simple algorithm, thus, cannot be bounded by a function of the network size. Moreover, it has been proven that under a certain variant of the branching-program model [4, 3], one cannot have a bounded-complexity algorithm for general synchronization which gives optimal synchronization bounds [19].

3 Optimal Synchronization With Drifting Clocks

In this section we present and analyze a new external clock synchronization algorithm. The algorithm is general and optimal, and thus not always efficient. However, the complexity of the algorithm is expressed in terms which are bounded by a polynomial in the size of the system in most practical scenarios, as we discuss in Section 4.

We construct the new algorithm in two steps. We first reduce the problem to a pure dynamic graph problem, using the Clock Synchronization Theorem, and then give an algorithm for the graph problem.

Our starting point is the general algorithm sketched in Section 2.3. Fix an external synchronization system. By Theorem 2.1, we know that the output of an optimal clock synchronization algorithm at point p is essentially the distance between p and the source point in the synchronization graph associated with the view of the execution from p. Thus, we can restate the problem as computing, at every point p, the distances between p and the source point. We can further abstract the problem as the following more general dynamic graph problem.

Accumulated Graph Distance Problem (AGDP).

- Initially, there is a graph with exactly one node marked as *source*. The source is marked *live*.
- Thereafter, the input is given in steps. In each step a new node and a few edges are added to the graph. The new node is marked *live*, and the new edges connect only other live nodes to the new node. At the end of each step, some of the nodes at the endpoints of new edges are unmarked as *live* (they will be called "dead" later).
- The task is to compute, at all times, the distance from the source point to each live point.

The central concept we need for the reduction of external synchronization to AGDP is formalized in the following definition.

Definition 3.1 *Let β be a view. The* live points *of β are all points p such that either*

- *p is the last point at some processor v, or*
- *p is a send event of a message whose receive event is not in β.*

3.1 Transforming External Synchronization to AGDP

It is straightforward to reduce the external synchronization problem to AGDP as follows (see Figure 2 for pseudo-code). As mentioned above, the local view of an execution can be thought of as a graph. As the execution unfolds, the local view is extended by adding nodes (events) and edges (e.g., message deliveries). Suppose for the moment that somehow, all events in the local view from a point are reported to the processor at or before that point (this is accomplished using a variant of standard techniques as we explain shortly). In cases where the view is extended by more than just a single node and its incident edges (this may occur

```
State:
  $H_v$:        set of events with local times and location of occurrence        history buffer
  $C_{vu}[w]$:  for each neighbor $u$ and processor $w$, a pointer to $H_v$       $C_{vu}[w]$ is last
                                                                 event in $w$ $v$ knows $u$ knows
  $M$:          set of events                                                     current message

Event: message $M$ sent to a neighbor $u$
  $M \leftarrow \{p \in H_v \mid LT(p) > C_{vu}[\mathrm{loc}(p)]\}$                fill message contents
  for each $w \in V$:
    $C_{vu}[w] = \max\{LT(p) \mid p \in H_v \text{ and } \mathrm{loc}(p) = w\}$
  $H_v \leftarrow \{p \in H_v \mid \text{ for some neighbor } u' \; LT(p) \le C_{vu'}[\mathrm{loc}(p)]\}$      garbage-collect

Event: message $M$ received from a neighbor $u$
  $H_v \leftarrow H_v \cup M$
  for each $w \in V$:
    $C_{vu}[w] = \max\{LT(p) \mid p \in H_v \text{ and } \mathrm{loc}(p) = w\}$
  $H_v \leftarrow \{p \in H_v \mid \text{ for some neighbor } u' \; LT(p) \le C_{vu'}[\mathrm{loc}(p)]\}$      garbage collect
  $M \leftarrow \emptyset$                                                        processing done
```

Figure 2: *Code for the Transformation Algorithm*

when a message is received), we break the insertion into a sequence of insertions. Nodes are marked live, and unmarked, using Definition 3.1. The edge weights are computed according to Definition 2.1.

To complete the reduction, we describe a way to ensure that at any point p, all events in the local view from p are reported to the processor in which p occurs, by the time p occurs. This is done using a technique similar to the "vector clocks" algorithm, used for asynchronous systems [14, 9, 23]. Each processor v maintains, for each neighbor u, an array C_{vu} with an entry $C_{vu}[w]$ for each processor w in the system. Intuitively, $C_{vu}[w]$ indicates the last event in w which v knows that u knows: more precisely, $C_{vu}[w]$ is the last event in w which was either reported to u by v or or reported to v by u. In addition, each node v maintains a local history buffer H_v which records all known events p in the system such that for some u, $LT(p) > C_{vu}[w]$, where $w = \mathrm{loc}(p)$ is the processor in which p occurred. When a message is sent by v to a neighbor u, all events that v does not know that u knows, i.e,. all events in the set $\{p \in H \mid LT(p) > C_{vu}[\mathrm{loc}(p)]\}$ are sent; the array C_{vu} is updated accordingly, and some of the events in H_v may be discarded. When a message arrives from a neighbor u, the events reported in the message are used to update H_v. They are also used to update C_{vu}: the new value of $C_{vu}[w]$, for each $w \in V$, is the maximum (with respect to time) of the old value of

$C_{vu}[w]$ and the last event in w reported in the new message from u.

We now analyze the transformation algorithm above.

Lemma 3.1 *Let p be any point in an execution of the system, and suppose that p occurs at a processor v. Then the set of events reported to v up to point p is exactly the local view of the execution from p.*

Proof Sketch: The lemma is proven using the following stronger invariant (recall that "$\rightarrow$" denotes the happens-before relation).

> Let v be any processor, and let p be any point occurring at v. For each neighbor u of v, let $R_u(p)$ be the latest between the last send event from v to u and the last receive event of a message from u to v. Then at point p:
>
> - H_v contains all events q in the local view from p such that $q \rightarrow R_u(p)$ for some neighbor u.
> - $C_{vu}[w]$ is the last event which occurred at w such that $w \rightarrow R_u(p)$, for all nodes w and all neighbors u.

The invariant is proven by a straightforward induction on the steps of the execution. ▮

The following lemma is useful to amortize the communication overhead of the algorithm.

Lemma 3.2 *Each event is reported at most once over each edge in each direction.*

Proof: Follows from the fact that once an event which occurred at a processor w is reported in a message from a processor v to a processor u, $C_{vu}[w]$ is advanced so that the reported event will not be reported again on that link. ▮

The following lemma gives a bound on the space requirement under a certain assumption for the "relative system speed," i.e., the rate in which events occur in different processors in the system.

Lemma 3.3 *Suppose that the maximal number of events in the system between two successive send events on a link is at most K_1. Then the transformation algorithm above can be implemented using $O(K_1 D + \delta|V|)$ space at each node, where D is the diameter of the network, and δ is the number of neighbors of that node.*

Proof: The space required to implement the algorithm consists of two parts: the C arrays and the H buffer. Implementing the arrays C requires, at each processor, $\delta \cdot |V|$ pointers to the H buffer. We now prove that for all $v \in V$, $|H_v| \leq K_1(D+1)$, where $|H_v|$ denotes the number of events stored in H_v. Fix a time t. For each $w \in V$, let p_w be an event occurring at processor w such that $LT(p) \leq C_{vu}[w]$ for all neighbors u of v, and define $S_{vt} = \{p_w \mid w \in V\}$. S_{vt} is the set of all "oldest" events in H_v in the following sense: if an event p' occurring at processor w is added to H_v after t, then $LT(p') > LT(p_w)$. Using the assumptions of the lemma, we bound the number of events occurring in the system until all events in S_{vt} are purged from H_v. Fix a node $w \in V$, and let q_w be the event immediately following p_w at w. Consider a shortest path from w to v. By the assumptions, the report of q_w is progressing along this path and will therefore arrive at v after no more than $K_1 D$ events have occurred in the system. After no more than additional K_1 events, q_w is reported by v to all neighbors of v. Since at this point, $q_w \rightarrow C_{vu}[w]$ for all neighbors u of v, and since $p_w \rightarrow q_w$, p_w must have been purged from H_v. Thus at that point in time, all events in S_{vt} have been purged from H_v. Since at any point in time, H_v contains only events which occurred before that point, we may conclude that $|H_v| = O(K_1 D)$ always. ▮

Remark: bit and word complexity. The complexity analysis above is for the "word model" where we assume that a memory word is sufficient to contain a node identifier. In our case, a the natural node identifier is a pair consisting of processor ID and local time. In the case of infinite executions, the local time grows unboundedly. For node labels under infinite executions, it is possible to use the bound on the system relative speed to apply a garbage collection scheme which recycles node labels. Thus the bit-complexity of the algorithm can be bounded. Details will be provided in the final version of the paper.

Another subtle point in our analysis for the bit-complexity model is that edge weights are treated as real numbers. From the information theoretic point of view, the space complexity of a real number is infinite. We avoid this difficulty by assuming that we can store time-stamps in special slots in messages and in memory, and apply to them only linear transformations. (This is the model under which the space lower bound was proven [19].)

We remark that from the practical viewpoint, both difficulties above are non-problems: a time-stamp is represented by a fixed-length structure (e.g., 64 bits in NTP [16])

3.2 A Solution to AGDP

We now describe an algorithm solving AGDP (see Figurefig-agdp for pseudo-code). Intuitively, the AGDP specification allows edges to be linked only to live nodes; we shall see that dead nodes can be completely ignored, so long as we keep track of distances between live nodes. Specifically, the algorithm is as follows.

Algorithm for AGDP. Let , $= (V, E, w)$ be the graph defined by the input to AGDP. The idea of the algorithm is to maintain a directed weighted graph $G = (V', E', w')$ which is a succinct representation of , . Initially, $G =$, , i.e., G consists of a single node (the source node). In each input step, the new node is added to V', and the new edges are added to E' (we later prove that all live nodes of V are in V', so that edge insertion to G is well defined). Let H denote the graph G extended by the new node and edges. Next, construct a fully-connected graph whose nodes are the all nodes in H, where the weight of an edge is the distance between

```
State:
  $G$,                                          succinct representation of accumulated input
  $H$:      directed weighted graphs            intermediate graph for computation

Input step: insert node $p$, edges $F$, unmark nodes $Y$
  $V(H) \leftarrow V(G) \cup \{p\}$
  $E(H) \leftarrow V(G) \cup F$
  $w_H(e) \leftarrow \begin{cases} w_G(e), & \text{if } e \in E(G) \\ w_F(e), & \text{if } e \in F \end{cases}$
  $V(G) \leftarrow V(H) - Y$
  $E(G) \leftarrow V(G) \times V(G) - \{(p,p) \mid p \in V(G)\}$
  for each $(p,q) \in E(G)$
    $w_G(p,q) \leftarrow d_H(p,q)$
```

Figure 3: *Code for the AGDP Algorithm.*

the nodes in H: this is done by running a dynamic version of all-pairs shortest-paths algorithm on H, to be explained later. From the fully-connected graph we finally delete all dead nodes and their incident edges, to obtain the graph used by the algorithm in the next step.

We claim that the distance between any two live points in G is exactly the same as the distance between these points in the original graph. This claim is proven by the following invariant.

Lemma 3.4 *Let $G = (V', E', w')$ be the graph generated by the algorithm above for input describing , $= (V, E, w)$. Then:*

1. *V' is exactly the set of all non-dead nodes of V.*
2. *For each ordered pair (x, y) of non-dead nodes of V we have that $(x, y) \in E'$ and that $d_\Gamma(x, y) = w(x, y)$.*

Proof: By induction on the inputs. The base case is trivial, since initially, $G =$, and they both contain a single live node and no edges. Suppose now that a new node p is added to , with some incident edges. To fix notation, let , * denote the extended , , and G^* denote the the graph generated by the algorithm after the insertion. H will denote the intermediate graph computed before G^* is determined. We also use the following additional notation: for a path P, let $|P|$ denote its length, and let $\text{first}(P), \text{last}(P)$ denote its first and last node, respectively. Assertion (1) of the lemma follows directly from the induction hypothesis and the fact that H contains all nodes which are live in , *. We now prove Assertion (2). Let x, y be any pair of nodes in G^*. We prove first that $w_{G^*}(x, y) \geq d_{\Gamma^*}(x, y)$, and then the reverse inequality.

By construction, there exists an edge (x, y) in G^*, which corresponds to a shortest path P in H. Since P is a shortest path, it is simple, and hence we may decompose P into three segments $P = P_0P_1P_2$ where:

- P_1 contains all edges incident to the new node p,
- P_0 is the prefix of P up to P_1, and
- P_2 is the suffix of P from P_1.

(Each of the segments may be empty.) Clearly, only P_1 can contain new edges. Hence P_1 appears in , *, while P_0 and P_2 consist of edges of G only. Furthermore, we claim that each of P_0 and P_2 consist of one edge, if it exists. This follows from the fact that P_0 (and similarly P_2) is a shortest path between its endpoints, and by the induction hypothesis. Next, we consider the path Q in , * obtained by concatenating the path corresponding to P_0 (whose existence is guaranteed by the induction hypothesis), then P_1, and then the path corresponding to P_2 (which, again, exists by induction). Since Q is a path in , *, and since $|Q| = |P| = w_{G^*}(x, y)$, we may conclude that $w_{G^*}(x, y) \geq d_{\Gamma^*}(x, y)$.

We now turn to prove that $w_{G^*}(x, y) \leq d_{\Gamma^*}(x, y)$. Let Q be a shortest path between two non-dead nodes x, y in , *. We decompose $Q = Q_0Q_1Q_2$ as above, i.e., only Q_1 may contain edges incident to the new node p. By the algorithm, Q_1 is also a path in H. Furthermore, since Q_0 and Q_2 are shortest paths in , , we have by induction that there exist edges $e_0 = (x, \text{last}(Q_0))$ and $e_2 = (\text{first}(Q_2), y)$ in G such that $w_G(e_0) = |Q_0|$ and $w_G(e_2) = |Q_2|$. It follows that the path P obtained by concatenating e_0, Q_1 and e_2 is a path in H and that $|Q| = |P|$. Therefore, $d_{\Gamma^*}(x, y) = |Q| \geq d_H(x, y) = w_{G^*}(x, y)$. ▮

We summarize the properties of the AGDP algorithm in the following lemma.

Lemma 3.5 *Suppose that the number of live points in an instance of AGDP is always smaller than L, for some given L. Then AGDP can be solved in space $O(L^2)$ and time $O(L^2)$ per edge insert operation.*

Proof: The space bound is immediate from the fact that the number of nodes in G and H is always $O(L)$. The time bound follows from a simple observation due to Ausiello et al. [2]: Whenever an edge (p, q) is inserted, the distance function can be updated by comparing, for pair of nodes r, s, $d(r, s)$ to $d(r, p) + w(p, q) + d(q, s)$, where d denotes the old distance function. ▮

We conclude with the following theorem, which is the main result of this paper.

Theorem 3.6 *Suppose that in all executions of some external clock synchronization system of diameter D, the number of live points in a local view of a processor never exceeds L, and that between any two events on any processor, there are at most K_1 events in the system. Then the algorithm specified above is an optimal external synchronization algorithm with space complexity* *at most $O(L^2 + K_1D)$, time complexity at most $O(L^2)$ per message, and message size bounded by $O(K_1D + \delta|V|)$, where δ is the maximum degree in the system.*

3.3 Dealing with Message Loss

Message loss presents a few problems for the algorithm above. For the AGDP algorithm, the send events of lost messages may be considered as live points indefinitely. The only way to avoid that is to assume the existence of some detection mechanism which eventually flags messages as lost, thus allowing us to mark the corresponding point as not live. Another problem arises for Lemma 3.3, since with arbitrary message loss, we cannot know how many events will occur until a report reaches some node. One way to overcome this problem is to have a refined assumption, stating that the number of event in the system between two consecutive successful message *deliveries* is no more than K_1.

4 Applications

The complexity of the algorithm specified in Section 3 depends on the number of live points in the system. The following simple lemma bounds this number in most practical systems.

Lemma 4.1 *Suppose that in all executions of a given connected systems, there are at most K_2 messages sent over a link in one direction between two consecutive message sends in the other direction. Then the number of live points in any local view in an execution is $O(K_2|E|)$.*

Combining Lemma 4.1 with Theorem 3.6, we get the following useful corollary.

Corollary 4.1.1 *Suppose that in all executions of a given system with diameter D, between any two events on any processor, there are at most K_1 events in the system, and that between any two consecutive sends events in one direction on a link, there are at most K_2 send events in the other direction of that link. Then there exists an optimal external clock synchronization algorithm with space complexity at most $O((K_2|E|)^2 + K_1D)$, time complexity at most $O((K_2|E|)^2)$ per message, and message size bounded by $O(K_1D + |V|^2)$*

Using Corollary 4.1.1 we can derive complexity bounds on most synchronization systems. We demonstrate with two prominent examples: NTP and Probabilistic Clock Synchronization.

The infra-structure of NTP [15, 16], used for clock synchronization over the Internet, consists of a distributed system of *time servers*. Time servers connected to a radio clock or to an atomic clock are declared to be *level* 0 *servers*; other servers obtain their time from (usually more than one) lower level servers, and their level is essentially the number of hops to the closest server. A client gets a time estimate by consulting one (or more) of the servers. Using the terminology of Section 2, one can model this situation by assuming an abstract source node representing standard time, connected to level 0 servers with links representing the accuracy of those servers;

the rest of the network is represented by its underlying graph. NTP uses remote procedure call mechanism for communication. The servers are probed periodically, where the length of the period is C minutes, where $1 \leq C \leq 16$. This organization has the property that the number of events in the system between two consecutive sends on a link is linear in the number of nodes, and the number of live points in any given time are linear in the number of edges. More specifically, in the language of Corollary 4.1.1, we have that $K_1 \leq 16|V|$ and $K_2 \leq 2$. We can therefore conclude that that under the communication pattern of NTP systems, the space complexity of the new algorithm is $O(|E|^2)$.

Probabilistic clock synchronization [5] is founded on a different idea. The basic observation is that the behavior of a link can be roughly described as obeying a probability distribution under which a quick round-trip is likely to occur within a few trials. When a client in a typical system notices that its synchronization bounds have become too loose (due to clock drift with the passage of time), it will initiate a burst of round-trip probes until it can deduce sufficiently tight synchronization bounds. Probabilistic clock synchronization is not very detailed about the organization of the servers for more than a two-level hierarchy: we shall assume that they are organized in a similar fashion to NTP. Let us now do a crude analysis of such a system. In line with the basic assumptions underlying probabilistic clock synchronization, let us denote by X_e a random variable which take the value 1 when a successful (i.e., quick) round trip is made on a link e. Let us further denote by $Y_e(t)$ the random variable which takes the time duration of a succession of trials starting at time t until $X_e = 1$. We assume that the Y_e are mutually independent, and that for each e, $\Pr(Y_e \leq T) \geq p_0$ for some parameters T and p_0. We further assume that for all times t, the probability that a processor v loses synchronization (and hence starts a series of trials) is at most p_1, for some parameter $p_1 \ll p0$. Under these simplistic assumptions, we can apply Corollary 4.1.1 with $K_1 = O(p_1|V|T)$ and $K_2 = 2$ and conclude that with high probability (polynomial in p_2), the space and time complexities of our algorithm are $O(|E|^2)$.

Acknowledgment

The second author would like to thank Sergio Rajsbaum for many very valuable and enjoyable discussions.

References

[1] H. Attiya, A. Herzberg, and S. Rajsbaum. Optimal clock synchronization under different delay assumptions. *SIAM J. Comput.*, 25(2):369–389, Apr. 1996.

[2] G. Ausiello, G. F. Italiano, A. M. Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *J. of Algorithms*, 12(4):615–638, Dec. 1991.

[3] A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982.

[4] A. Borodin, M. Fischer, D. Kirkpatrick, N. Lynch, and M. Tompa. A time-space tradeoff for sorting on non-oblivious machines. *J. Comp. and Syst. Sci.*, 22:351–364, 1981.

[5] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[6] 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.

[7] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, July 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*, pages 284–293, Aug. 1994.

[9] C. Fidge. Logical time in distributed systems. *Computer*, 24(8):28–33, 1991.

[10] J. Y. Halpern, N. Megiddo, and A. A. Munshi. Optimal precision in the presence of uncertainty. *Journal of Complexity*, 1:170–196, 1985.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.

[12] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Computation*, 62(2-3):190–204, 1984.

[13] 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.

[14] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[15] D. L. Mills. Internet time synchronization: the Network Time Protocol. *IEEE Trans. Comm.*, 39(10):1482–1493, Oct. 1991.

[16] D. L. Mills. The Network Time Protocol (version 3): Specification, implementation and analysis. RFC 1305, Network Working Group, University of Delaware, Mar. 1992.

[17] Y. Moses and B. Bloom. Knowledge, timed precedence and clocks. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 294–303, 1994.

[18] J. Palsetia. Ananlysis of clock synchronization algorithms. Masters of science project, Detp. of Electrical and Computer Engineering, Northeastern University, May 1997.

[19] B. Patt-Shamir. *A Theory of Clock Synchronization.* PhD thesis, MIT Lab. for Computer Science, Oct. 1994.

[20] 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.

[21] F. B. Schneider. Understanding protocols for Byzantine clock synchronization. Research Report 87-859, Department of Computer Science, Cornell University, Aug. 1987.

[22] B. Simons, J. L. Welch, and N. Lynch. An overview of clock synchronization. Research Report RC 6505 (63306), IBM, 1988.

[23] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, Edmonton, AB, Canada, Aug. 1989. ACM Press.

[24] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *J. ACM*, 34(3):626–645, July 1987.