Virtual Time and Timeout in Client-Server Networks

Jayadev Misra*

July 13, 2011

Contents

*This work is partially supported by National Science Foundation grant CCF-0811536.

1 Introduction

This paper proposes that virtual time and virtual time-outs should be available as tools for programming distributed systems. Virtual time is already used for event ordering in distributed systems [9], though the numeric value of virtual time is irrelevant in this context (see Section 1.1.1; page 2). Virtual time-out, i.e., waiting for a specific amount of virtual time, has not been used in programming distributed systems. Both virtual time and time-outs are used in discrete event simulations though such an application is usually implemented on a single machine, rather than on distributed systems.

We propose to make virtual clock, i.e., virtual time and time-out, available in full generality over a distributed set of machines. We argue that the benefits extend beyond mere ordering of events or simulations. We show an example where independent threads may be executed in a certain order by making use of virtual clock. We show solution to a combinatorial example, computing shortest path in a graph, that can be structured as a set of concurrent threads operating with virtual time-outs, see Section 6 in page 23.

Virtual clocks have been implemented in a concurrent programming language, Orc [12, 8], designed by the author and his co-workers. We find that virtual clocks eliminate many routine book-keeping operations from explicit consideration by the programmer.

1.1 Background

1.1.1 Causal Model of Virtual Time

In a classic paper [9], Lamport introduced the clock synchronization problem across a set of machines. Lamport argued that real-time clocks of different machines can not be perfectly synchronized; therefore, determining the order of events across machines is not a well-defined problem. However, it is possible to implement virtual clocks at different machines and synchronize them so that if

an event at one machine causes an event at another machine (perhaps through a chain of events) then the preceding event happens earlier in *virtual time*[1].

The synchronization algorithm time-stamps each event with a natural number, its virtual time, so that an event that causally precedes another will be assigned a lower virtual time. For a set of machines that communicate through messages, (1) events occurring at a single machine are strictly ordered in virtual time according to the order of their execution, and (2) each message carries the time-stamp of the corresponding send event and the receive event is assigned a strictly higher time-stamp than the send event.

Lamport's algorithm and some of its variations [6, 10, 1, 15] have become fundamental in designs of practical distributed systems. Morgan [13] shows several distributed algorithms using virtual time.

The numeric value of virtual time is irrelevant in the causal model because the time-stamps of different events are merely compared. Therefore, doubling all virtual times or increasing all by some fixed amount would not affect these comparisons.

1.1.2 Simulation Model of Virtual Time

Discrete event simulations of physical systems employ virtual time to mimic real time in the physical system. Typically, a simulation is implemented on a single machine employing a single virtual clock. If event x happens at time t in the real world, it is *scheduled* to happen at virtual time t in the simulation; the scheduler adds the time and event pair (t, x) to an event queue. Whenever no event is being processed by the simulator, the pair with the smallest associated time, say (t', x'), is removed from the event queue, the virtual time is advanced to t', and event x' is processed. Processing of an event may update the state of the simulation, and it may also update the event queue by adding/removing pairs from it. A step in the real world, such as a customer receiving service for k units of time, is simulated as follows: when the start of service event is processed at virtual time t, it schedules the end of service event to be processed at virtual time $t + k$, and enters it into the event queue.

It follows that if event x precedes event y in real time, then x is processed before y in the simulator.

The numeric value of virtual time is important because entities in a physical system, e.g. persons waiting in a queue for service, wait for specific lengths of time. The computation required for processing of events consumes no virtual time; virtual time is advanced only when the processing of next event from the event queue is started.

Unlike the causal model which is meant for implementation on a distributed network of processes, a simulation is typically implemented on a single processor; so, there is a single virtual clock under consideration in a simulation. Distributed simulations [] use multiple virtual clocks, but a client merely processes a single event at any moment and then waits for a communication.

[1] Lamport used the term "logical time". We use "virtual time" to denote logical time whose magnitude is also relevant.

1.2 Contributions of this Paper

We propose facilities for accessing the value of virtual time and waiting for a specific amount of virtual time in distributed systems. Our model of virtual time combines both the causal and simulation models, and generalizes them. We show a distributed algorithm for implementation of virtual time.

We regard a distributed system as a client-server network; clients communicate with each other through the servers. Each client has a virtual clock. Each step of a client consumes some amount of virtual time within given lower and upper bounds, as specified for that step. A client may wait for k units of virtual time by executing a step $Vwait(k)$, thus implementing virtual time-out.

Each event in a computation is assigned a virtual time of occurrence, called its *time-stamp*, and the time-stamps have to obey a number of safety conditions, given in Section 3, page 7. The safety conditions include the condition on causally dependent events, as in the causal model. And, they also include constraints on the virtual time consumed by a step, specified by a lower and an upper bound on the virtual duration of the step.

The time-stamping algorithm requires each client to maintain a virtual clock and the servers to help synchronize the virtual clocks of the clients.

2 Computation Model

In this section, we describe the computation model of client-server network, first informally and then formally, without virtual time. We introduce virtual time in Section 3.

2.1 Informal Model

A client-server network consists of a finite number of clients and servers. Each client runs its own program and interacts with servers by making *calls* on them. A server may respond to a client's call by returning some data, called its response. A server does not respond as long as it does not have or can not compute the appropriate data; consequently, a server may not respond at all.

Client A client executes *steps* where the *start* and *end* of each step are *events*. A *trace* of a client, denoting a possible execution of the client, is a finite sequence of events. It is possible for a step to start but never end in an execution; in that case, only its start event is included in the trace. The steps of a client may be concurrent. Therefore, the start and end events of various steps may be interleaved arbitrarily. Further, steps may be nested where one step may include others as substeps. We make no assumption about the nature of steps, except the ones noted below.

Certain steps are designated as *communication* steps. A communication step at a client starts by calling a server and ends on receiving the server response. Correspondingly, a communication step at a server starts by receiving a client call and ends by sending a response.

A special client step $Vwait(k)$, where k is a non-negative integer, behaves as a skip, i.e., it has no effect on the program state[2]. Its semantics in terms of virtual time is described in Section 3, page 7.

Server A server accepts requests from clients and responds to them, if possible. We make no assumption about the nature of a server. A server could be as trivial as one implementing the identity function that merely returns the argument of the call as its response. Or, it could be as elaborate as a web search engine, a database, a shared memory manager, a semaphore or a message communicating channel, or, even, a real-time clock (see Section 4.4.5, page 20).

As an example, consider a server that implements an asynchronous unbounded channel for message communication between clients. It may support two kinds of calls: (1) to put an item in the channel, and (2) to get an item from the channel. A put step would normally terminate, and then the server sends an acknowledgement to the calling client. A get step is blocked until there is an item in the channel; it may be blocked forever if the channel remains empty. There is no special treatment in our theory for message communication; messages are sent and received by calling a channel server. Clients may also exchange data by communicating with a server that implements a shared memory, or a database, for example.

A response from a server depends not only on the corresponding call but, possibly, on other calls. For example, in a server that implements an unbounded channel, a put call immediately elicits a response in the form of an acknowledgement; the response depends only on the call. However, a get call's response depends not only on this call but also on the call that put the item in the channel in the first place. The exact form of dependence is specific to each server.

Note: A server does not make any calls in this model. A more general model of the network is a multi-layer graph where a pure client is a node in the top layer, a pure server a node in the bottom layer, and a node may call any node at a lower layer. Our approach can be extended to this general model.

2.2 Formal Model

2.2.1 Client and Server

Associated with each entity, i.e., a client or a server, is (1) a set of *steps*, (2) a binary relation, called *precedes* and denoted by $\prec$, and (3) a set of *traces*.

Step The set of steps of different entities are disjoint. Each step has an associated *start* and *end* event. Certain steps are designated as *communication* steps. The start event of a communication step at a client is designated *client-call*, the corresponding end event, receiving the response from the server, is *client-response*. Similarly, the start event of a communication step at a server is *server-call* and the end event is *server-response*.

[2]Theoretically, $Vwait(k)$ is a different step for each value of k.

Precedes For events x and y at any entity (server or client), $x \prec y$ denotes that event x *precedes* y, and we say that x is a *precedent* of y. Relation $\prec$ is a strict partial order, i.e., for all events x, y and z, we have:

asymmetry: $\neg(x \prec y \ \wedge \ y \prec x)$ and,
transitivity: $(x \prec y \ \wedge \ y \prec z) \Rightarrow (x \prec z)$.

Event x is an *immediate precedent* of event z if $x \prec z$ and there is no event y such that $x \prec y \prec z$.

Every start event precedes the corresponding end event. For a communication step at a client whose start event is x and end event y: x is an immediate precedent of y, x is not an immediate precedent of any other event and y has no other immediate precedent. At a server, a server-call event has no precedent, but the server-response is preceded by the corresponding server-call event and, possibly, other events as well.

Trace A trace of an entity denotes a possible execution. It is a finite sequence of events that satisfies the following conditions. Below, T is a finite sequence of events, and x and y are events.

- (T1) Tx is a trace implies T is a trace and for all y, where $y \prec x$, $y \in T$.
- (T2) Txy is a trace and x does not precede y implies Ty is a trace.

Condition (T1) implies that traces are prefix-closed, i.e., all prefixes of a trace are also traces. Further, an event can occur in a trace only if all its precedents have already occurred. (An event x that can occur if *either* y or z has occurred can be modeled by having two distinct events x' and x'' that mimic x, and whose precedents are y and z, respectively.)

Condition (T2) says that if y can occur immediately after Tx but x is not a precedent of y, then y can occur without the occurrence of x. It may seem that the appropriate condition should be: if event x is not in trace T, but all its precedents are in T, then Tx is a trace. This is, however, a strong requirement. Calling that x is "enabled" in this situation, the stronger condition amounts to: an enabled event remains continuously enabled until it is executed. Condition (T2) permits disabling an enabled event through occurrence of other events; it merely says that an event y that has been enabled but not disabled may be executed. Henceforth, we assume that each entity has at least one trace. Hence, $\langle\ \rangle$ is a trace of every entity, from prefix-closure.

2.2.2 Network

A client-server network, or just *network*, consists of a finite number of entities, and all communications are between the clients and servers included in the network. So, a network is a client none of whose steps is a communication step.

Step The set of steps of a network is the union of the steps of its component entities. Recall that the steps of distinct entities are disjoint. The events of a network are the start and end events of the steps.

Precedes The precedes relation of a network is a strict partial order that includes the precedes relation of every component entity and a set of communication relations as described below. A communication relation is of the form $\{(x, x'), (y', y)\}$, where x and y are the start and end events of a client communication step, and x' and y' those of a server communication step. Any communication step, of a client or server, may participate in at most one communication relation.

Trace Let T be a finite sequence of events of the component entities of a network. The *projection* of T on u, written as T_u, is the subsequence of events of T that are from entity u. Sequence T is a network trace if and only if: (1) T_u is a trace of u for every entity u in the network, and (2) for any event x in T all precedents of x come before x in T.

Lemma 1 Network traces satisfy the conditions on traces, i.e.,

1. Tx is a network trace implies T is a network trace, and for all y, where $y \prec x$, $y \in T$.
2. Txy is a network trace and Ty is not a network trace implies $x \prec y$.

Proof: See Appendix A, page 26. □

Observe that network traces are prefix-closed, from condition (1) in Lemma 1.

3 Virtual Time

We augment the model of Section 2 by permitting each client to have a virtual clock that displays the current virtual time at that client; clocks of different clients may display different times.

Each step of a client has associated lower and upper bounds on the amount of virtual time the step consumes, where the lower bound does not exceed the upper bound. The bounds are natural numbers and, possibly, ∞ for the upper bound. For *Vwait*(k), both lower and upper bound are k, i.e., the step consumes exactly k units of virtual time. The bounds on steps are specified as part of the model of computation. Servers do not have virtual clocks nor *Vwait* as a step.

Typical virtual time bounds are (lower and upper bound given as a pair): (i) $(0,\ \infty)$ indicating that a step's virtual time characteristics are irrelevant, (ii) $(0,\ 0)$ indicating that a step consumes no virtual time, as is the case for a statistics collection step in simulation, for example, and (iii) $(k,\ k)$ for a virtual time-out step.

A *time-stamped*, or *timed*, network trace is a network trace in which every event x has an associated natural number v_x, called its *time-stamp*. A timed network trace T is *safe* if it meets the following *safety* conditions.

Safety Conditions

1. (Causality) Time stamps respect precedence, i.e., for events x and y in T, $x \prec y \Rightarrow v_x \leq v_y$.

2. (Monotonicity) Let x and y be events of a single client and x comes before y in T. Then, $v_x \leq v_y$.

3. (Duration) Let p be a client step whose start event x and end event y are both included in T. Let lb_p and ub_p be the lower and upper bounds on virtual time consumed by step p. Then, $lb_p \leq v_y - v_x \leq ub_p$.
   Further, if x is included T but y is not, then for every event z of this client in T, $v_z - v_x \leq ub_p$.

4. (Eagerness) Let x be a start event of a client in T. Then, $v_x = \max\{v_y |\ y \prec x\}$.

Convention: $\max(\{\}) = 0$ and $\min(\{\}) = \infty$.
For natural number n, $n < \infty$ always holds and $n + \infty$ is ∞.

Causality is the most elementary condition for virtual time. Monotonicity is a stronger requirement for clients; it implies causality for clients only. These are the only conditions imposed on virtual time in Lamport's causality model [9].

Duration condition says that the time stamps obey the timing constraints for steps at the clients. If both start event x and end event y of a step are included in a trace, their time-stamps are such that the duration of the step, given by $v_y - v_x$, is within the time-bounds specified for the step. If the trace contains only x but not y, then extending the trace by y should remain a possibility, i.e., the time interval from x to any event of the client in the trace does not exceed the upper bound of the step.

Eagerness condition is inspired by discrete event simulation where a step is started as soon as possible in virtual time.

The only condition that applies to server events is causality. In particular, the time stamps in a server trace may not be monotonic. The duration condition does not apply to servers because server steps have no associated bounds on virtual time they consume. A server is not required to start a step as soon as possible, as required by eagerness.

Example 1 Consider two concurrent threads in a client that attempt executing steps p and q in parallel. The time bounds for p is (1,1) and for q is (2,3). Let p denote the start event of p and p' its end event; similarly for q. In Table 1, we show several possible traces and assignments of time-stamps to them (time-stamp of an event is written below the event name). Traces 2 and 3 show different safe time-stamps for the same untimed trace. The last three traces have time stamps that violate some of the safety conditions. Trace 6 has no possible safe time-stamp.

Trace					Violates
1	p	q	p'	q'	-
	0	0	1	2	
2	q	p	p'	q'	-
	0	0	1	2	
3	q	p	p'	q'	-
	0	0	1	3	
4	p	q	q'	p'	Monotonicity
	0	0	2	1	
5	p	q	q'	p'	Duration for p
	0	0	2	2	
6	p	p'	q	q'	Eagerness for q
	0	1	1	3	

Table 1: Time-stamped traces

4 Implementation

We describe an algorithm that assigns time-stamps to events during an execution of the network. An execution of a network corresponds to a safe timed-trace; initially, the execution corresponds to the empty trace, which is safe, and each occurrence of an event has to be time-stamped so that all the safety conditions are met. The implementation does not have access to the trace sets of the entities; so it can not be based on an analysis of the possible future events.

We designate a client step to be *pending* during an execution if its start event has been time-stamped, but not its end event. A pending step is either *active* or *passive*; an active step is still executing and a passive step has completed its execution though its end event is still not time-stamped.

For a pending step, its *lower-deadline* is the smallest time-stamp that can be assigned to its end event; similarly, the *upper-deadline* is the largest time-stamp for its end event. Henceforth, for step p, we write ld_p and ud_p for its lower and upper-deadlines, and lb_p and ub_p for its lower and upper-bounds. When the start event of a step is time-stamped vt, deadlines ld_p and ud_p are set to $vt + lb_p$ and $vt + ub_p$, respectively. Subsequently, the lower-deadline may be revised to a higher value if the step is a communication and the server response indicates that the step can not be completed until some time after the current lower-deadline. The upper-deadline remains fixed.

A client's trace is *quiescent* if it has no extension by a start event. A client is quiescent in an execution if the corresponding trace is quiescent; that is, it can not start any step. An execution is quiescent if the corresponding trace is.

4.1 Time-stamping Algorithm

Initially Each client has a local variable vt that holds the current value of its virtual time. Initial value of vt is 0 at every client.

Server

(S) On executing server event x, set
$v_x := \max\{v_y |\ y \prec x\}$.

Client

(C1) On executing start event of step p:
Designate p passive if p is *Vwait*, otherwise it is *active*.
Set $v_x := vt$, where x is the start event of p.
Set $ld_p, ud_p := vt + lb_p, vt + ub_p$.

(C2) On executing end event of active step p:
Designate p passive.
If p is a communication step, set $ld_p := \max(ld_p, t)$,
where t is the time-stamp of the corresponding server-response.

(C3) At quiescence:
Let μ be the minimum over upper-deadlines of all pending steps.
If $ld_p \leq \mu$ for some passive step p, then
set $vt := \max(ld_p, vt)$; $v_y := vt$, where y is the end event of p. □

Remarks on the Algorithm

1. The algorithm does not specify how exactly a server should execute action (S). For server-call x', whose unique immediate precedent is client-call x, this amounts to setting $v_{x'}$ to v_x. Value v_x can be appended to the call itself, and the server can then acquire that value. Time-stamp of every other server event can be computed locally at the server. Similarly, the time-stamp of server-response can be appended to the response so that the client has the value of t in action (C2).

2. The algorithm is written in a non-deterministic style for the client, because it permits a number of possible implementations in practice. Actions (C1) and (C2) are executed when a step starts and completes, respectively; however, the choice of a step is non-deterministic if several possible steps qualify for execution. Action (C3) is executed at quiescence. The choice of the passive step p in (C3) is arbitrary as long as the associated condition is met. This is the only step in which vt may be updated (increased).

3. Suppose at quiescence the associated condition in (C3), $ld_p \leq \mu$, is not met by any passive step p. There are three possible scenarios then.

   (a) There is no pending step: client's execution has reached termination.

   (b) There is a pending passive step, but no active step: since there is no active step and all passive steps p fail to satisfy $ld_p \leq \mu$, this situation will persist. The client's execution can not proceed any further and the pending steps will remain pending, akin to a deadlock.

(c) There is a pending active step: the client has to wait to execute action (C2) next, so that a pending step may become passive, and, possibly, satisfy the condition in (C2).

4. Action (C3) can be executed only at quiescence just to implement the eagerness condition, as will become evident in the correctness proofs; see Section 4.2, page 11. If eagerness is not a concern, then (C3) may be executed at any point during an execution.

5. The algorithm may time-stamp end events in a different order from their completion order. Given pending steps p and q, p may become passive first, and then q, whereas q's end event may get time-stamped before p's. In this case, p's end event does not precede q's; so, from Section 2.2, Trace condition (T2), it is acceptable to shuffle their order.

6. Consider a passive step p for which $ld_p > ud_p$; this could happen if in action (C2), $t > ud_p$, so that ld_p gets set to t. Then the step remains passive forever because $ld_p > ud_p \geq \mu$, and it never satisfies the condition in (C3). Further, from Proposition 1 below, $vt \leq \mu$; so, $vt \leq ud_p < ld_p$. Thus, the virtual time will remain below ld_p, possibly, blocking the execution.

4.2 Correctness of the Implementation

We prove that the time-stamping algorithm meets the safety conditions of Section 3, page 8. Recall that μ is the minimum over upper-deadlines of all pending steps.

4.2.1 Preliminary Results

Proposition 1 For any client, $vt \leq \mu$ is invariant.

Proof: Initially, there is no pending step, so $\mu = \infty$. Further, every vt is 0. So, the proposition holds vacuously. Whenever a new upper-deadline, ud_p, is set in action (C1), it is of the form $vt + ub_p$, where $ub_p \geq 0$; so, the invariant is preserved. Whenever vt is updated in action (C3), it is by the assignment

$$vt := \max(ld_p, vt), \text{ where } ld_p \leq \mu$$

So, the invariant is preserved. □

Proposition 2 For step p with x and y as start and end events, the following invariants hold whenever v_x and/or v_y are defined.

$$v_x + lb_p \leq ld_p \quad \text{(I1)}$$
$$v_x + ub_p = ud_p \quad \text{(I2)}$$
$$ld_p \leq v_y \leq ud_p \quad \text{(I3)}$$

Proof of (I1): v_x is set just once in (C1) and never changed, and ld_p is then set to $v_x + lb_p$, thus satisfying (I1). The value of ld_p may subsequently change in (C2) by the assignment $ld_p := \max(ld_p, t)$, so it can only increase. Therefore, $v_x + lb_p \leq ld_p$ is maintained.

Proof of (I2): ud_p is set to $v_x + ub_p$ when v_x is set in (C1), and neither variable is changed thereafter.

Proof of (I3): We set v_y only in (C3). The precondition of this action is $ld_p \leq \mu$. We have to establish the post-condition $ld_p \leq v_y \leq ud_p$.

$$
\begin{array}{ll}
 & \{ld_p \leq \mu\}\ vt := \max(ld_p, vt);\ v_y := vt\ \{ld_p \leq v_y \leq ud_p\} \\
\equiv & \{\text{Using the axiom of assignment}\} \\
 & \quad ld_p \leq \mu\ \Rightarrow\ ld_p \leq \max(ld_p, vt) \leq ud_p \\
\equiv & \{ld_p \leq \max(ld_p, vt) \text{ always holds}\} \\
 & \quad ld_p \leq \mu\ \Rightarrow\ \max(ld_p, vt) \leq ud_p \\
\equiv & \{\max(ld_p, vt) \leq ud_p\ \equiv\ (ld_p \leq ud_p \wedge vt \leq ud_p)\} \\
 & \quad ld_p \leq \mu\ \Rightarrow\ (ld_p \leq ud_p \wedge vt \leq ud_p) \\
\equiv & \{\text{from Proposition 1, } vt \leq \mu \text{ and, } \mu \leq ud_p.\ \text{So, } vt \leq ud_p\} \\
 & \quad ld_p \leq \mu\ \Rightarrow\ ld_p \leq ud_p \\
\equiv & \{\text{by definition, } \mu \leq ud_p\} \\
 & \quad \mathit{true}
\end{array}
$$

□

Proposition 3 For any client, vt is monotonic.

Proof: The only assignment to vt is $vt := \max(ld_p, vt)$ in action (C3). □

Proposition 4 For any event z of a client, $v_z \leq vt$.

Proof: v_z is set to vt, and vt is monotonic from Proposition 3. □

4.2.2 Proofs of Safety Conditions

Theorem 1 Monotonicity condition holds in every timed trace.

Proof: All client events, both start and end, are time-stamped vt. Since start event comes before the end event and vt is monotonic (from Proposition 3), the result follows. □

Theorem 2 Causality condition holds in every timed trace.

Proof: Let x and y be events in a trace such that $x \prec y$. It is sufficient to prove the result when (1) both events are from one entity (a client or a server), (2) x is a client-call and y the corresponding server-call, and (3) x is a client-response and y the corresponding server-response.

(1) If both events are at a server, then from $x \prec y$, event y is not a server-call event because a server-call event has no precedent at a server. Hence, from action (S), $v_y = \max\{v_x |\ x \prec y\}$. So, $v_x \leq v_y$. If both events are at a client, x comes before y in the client trace, and from Theorem 1, $v_x \leq v_y$.

(2) Suppose x is a client-call and y the corresponding server-call. Then, from action (S), $v_x = v_y$.

(3) Suppose y is a client-response and x the corresponding server-response; suppose y is the end event of step p. In action (C2), we set $ld_p := \max(ld_p, t)$ where $t = v_x$. Hence, $v_x \leq ld_p$ following the assignment. From invariant (I3) in Proposition 2, page 11, $ld_p \leq v_y$. Therefore, $v_x \leq v_y$. □

Theorem 3 Duration condition holds in every timed trace.

Proof: Let step p have start and end events x and y in a timed trace. We show below that $lb_p \leq v_y - v_x$ (in the left) and $v_y - v_x \leq ub_p$ (in the right).

$v_x + lb_p \leq ld_p$	, from (I1)	$v_y \leq ud_p$	, from (I3)
$ld_p \leq v_y$	, from (I3)	$ud_p = v_x + ub_p$	, from (I2)
$v_x + lb_p \leq v_y$	, from above two	$v_y \leq v_x + ub_p$	, from above two
$lb_p \leq v_y - v_x$	, rewriting	$v_y - v_x \leq ub_p$	, rewriting

For step p whose start event x is in the trace but not its end event, we need to show that $v_z - v_x \leq ub_p$, for any event z of this client in the trace. From Proposition 1, page 11,

$$
\begin{array}{ll}
 & vt \leq \mu \\
\Rightarrow & \{v_z \leq vt, \text{ from Proposition 4, and } \mu \leq ud_p \text{ from the definition of } \mu\} \\
 & v_z \leq ud_p \\
\Rightarrow & \{ud_p = v_x + ub_p, \text{ from (I2) of Proposition 2, page 11}\} \\
 & v_z \leq v_x + ub_p \\
\Rightarrow & \{\text{rewriting}\} \\
 & v_z - v_x \leq ub_p
\end{array}
$$

□

We prove a lemma in order to prove the eagerness condition.

Lemma 2 Let TUz be a trace of a client where T is quiescent and z is a start event. Then U includes a precedent of z.

Proof: We use property (T2) of traces, from Section 2.2, page 5, which we write in an equivalent form: Txy is a trace and Ty is not a trace implies $x \prec y$.

Suppose that U includes no precedent of z. By induction on the length of U it can be shown that Tz is a trace. Since z is a start event, this means that T is not quiescent, a contradiction. □

Theorem 4 Eagerness condition holds in every timed trace.

Proof: Let z be start event of a client in the timed trace. We show that $v_z = \max\{v_x |\ x \prec z\}$.

Case 1) $v_z = 0$: All precedents of z come before it in the client trace, from the computation model of Section 2.2, page 5. From monotonicity, all events that come before z have time-stamp 0. Hence, $v_z = \max\{v_x |\ x \prec z\}$. Note that the result holds even when z has no precedent, because $\max(\{\}) = 0$.

Case 2) $v_z > 0$: Since the initial time stamp at the client is 0, the trace including z can be written in the form TUz where all events in U have time stamp v_z, and all events in T have strictly lower time-stamps. Therefore, vt increases following T. From the description of the algorithm, vt increases only at quiescence in action (C3); hence, T is quiescent. Further, z is a start event. From Lemma 2, page 13, U includes a precedent of z. Since all events in U have time stamp v_z, some precedent of z has time-stamp v_z; therefore, $\max\{v_x |\ x \prec z\} = v_z$. □

4.3 Progress Condition

We have shown in Section 4.2.2 that the time-stamping algorithm of Section 4.1 meets all the safety conditions. That, however, is not sufficient, because a time-stamping mechanism can meet all the safety conditions by doing nothing. The only trace corresponding to such an execution is the empty trace, which satisfies all the safety conditions. Our time-stamping algorithm also establishes a strong progress property, which we explore in this section.

The strongest possible progress condition is that an execution be extended if it is possible to extend it. In terms of the formal model, this amounts to:

- (Progress) Let the execution of the network correspond to the timed trace T. If T has a safe extension, then the execution will be extended.

Observe that T may have many safe extensions, and the execution will be extended corresponding to one of them.

Remarks The progress condition applies only if the *timed* trace corresponding to the current execution has an extension. It is possible that the untimed trace has an extension, but not the timed trace. To see this, consider a client that has two threads, one of which has a *put* step that puts an item in a channel and the other has a *get* step that retrieves the item. There is an untimed trace that includes end events of both steps. Suppose that the *put* step has lower-bound of 1 and the *get* step has both lower and upper bounds of 0. Then neither step can complete in the timed computation model. The *put* step can't be completed before virtual time 1, whereas *get* requires its response to be delivered by virtual time 0. Virtual time will never increase beyond 0 (because that will violate the duration condition on *get*), and hence *put* step will never be completed. So, the timed trace that includes the start events of both steps has no extension.

4.3.1 Proof of Progress Condition

Theorem 5 (Progress Theorem) The time-stamping algorithm implements the progress condition.

Proof: Let the timed trace corresponding to an execution be T, and T has a safe extension Tz. We show that the time-stamping algorithm will extend the execution. This holds under the weakest possible condition on the execution of

the algorithm, i.e., the algorithm is not permanently blocked if it is possible to execute any step, (S), (C1), (C2) or (C3). In the time-stamping algorithm, actions (S), (C1) and (C3) extend the execution by time-stamping events, whereas action (C2) merely updates the internal state without extending the execution. Note that (C2) can be executed only a finite number of times, at most once for each active pending step.

If z is a server event, then the execution can be extended by action (S).

Suppose T has an extension Tz where z is the start event of a client. Then the execution can be extended by action (C1), though it may assign a different time-stamp to z than the one in the extension.

If T has no extension by start event of any client, all client traces are quiescent. So, z is an end event, say of step q. Step q is passive or will become passive, since the trace can be extended by z. From (I3) of Proposition 2, page 11, $ld_q \leq v_z$. For any pending step p whose start event is x, from duration condition of Section 3, page 8, $v_z - v_x \leq ub_p$, or $v_z \leq v_x + ub_p = ud_p$. So, $ld_q \leq v_z \leq ud_p$, or $ld_q \leq ud_p$ for every pending step p. Hence, $ld_q \leq \mu$; therefore, action (C3) can be executed to extend the execution. □

4.3.2 A Sufficient Condition for Progress

The progress theorem is stated in terms of existence of extensions of timed-traces. For reasoning about the behaviors of programs in practice, we need simpler conditions that are based only the nature of steps, not traces. We should expect that a step that is not guaranteed to terminate —either because it is in an infinite computation, or because it depends on some step of another entity— should not impose a finite upper bound, because that may cause the virtual time to remain below its upper deadline should the step remain non-terminating. We formalize these notions in this section and prove a strong theorem about progress of execution.

Independent step A step in the untimed model is *independent* if it is guaranteed to complete. In particular, its completion is not conditional upon events that may or may not occur[3]. As examples, skip step is independent and so is a *Vwait* step, because in the untimed model it is equivalent to skip. A put step on an asynchronous unbounded channel is also independent. Conversely, a get step on such a channel is not independent because it has to depend on some put step for its completion. A step that is not independent is *dependent*. We now define these terms formally.

In the untimed model, a *terminal trace* of a network is either a trace that has no extension, or an infinite sequence of events each prefix of which is a trace. Step p, with start and events x and y, is *independent* if and only if the following conditions hold.

1. Any terminal trace that includes x also includes y. Consequently, a pending independent step is or will become passive.

[3]An independent step is also called "wait free" in distributed systems literature.

2. Any precedent of y is totally ordered with respect to x, i.e., if $z \prec y$ then either $z \prec x$, $z = x$ or $x \prec z$.

Requirement (1) guarantees that once p is started, it will be completed. Requirement (2) states that any event z required for completion of the step (its end event y) is either x itself, is a precedent of x, or is spawned by x. An event unrelated to the starting of the step can not influence its completion.

It may seem that any execution in which there is a pending independent step will be extended, because the termination of the step is guaranteed. This is not true because of the timing constraints; see example below.

Example 2 A client has two threads, one of which puts a value in a channel and the other gets the item. The *put* step is independent. The *put* step has lower-bound of 1 and the *get* step has both lower and upper bounds of 0. The *put* step can't be completed before virtual time 1, whereas *get* requires its response to be delivered by virtual time 0. The virtual time will never increase beyond 0 (because that will violate the duration condition on *get*), and hence *put* step will never be completed. An execution that includes only the start events of *put* and *get*, and thus has a pending independent step, will never be extended. □

Lemma 3 The server step corresponding to an independent communication step at a client is independent.

Proof: Let p be an independent communication client step and p' the corresponding server step. Let x and y be the start and end events of p, and x' and y' of p'. So, $x \prec x'$ and $y' \prec y$.

Any terminal trace that contains x' also contains x, because $x \prec x'$. From independence of p, the trace also contains y. Since $y' \prec y$, it also contains y'.

We show that for any z, if $z \prec y'$, then either $z \prec x'$, $z = x'$ or $x' \prec z$.

$$
\begin{array}{ll}
 & z \prec y' \\
\Rightarrow & \{y' \prec y\} \\
 & z \prec y \\
\Rightarrow & \{p \text{ is independent}\} \\
 & z \prec x,\ z = x \text{ or } x \prec z \\
\Rightarrow & \{z \prec x \text{ or } z = x \text{ implies } z \prec x', \text{ from } x \prec x'\} \\
 & z \prec x' \text{ or } x \prec z \\
\Rightarrow & \{x \text{ immediately precedes only } x'; \text{ so, } x \prec z \ \Rightarrow \ (z = x' \vee x' \prec z)\} \\
 & z \prec x' \text{ or } z = x' \text{ or } x' \prec z
\end{array}
$$

□

Lemma 4 Let x and y be start and end events of an independent communication server step in a timed trace. Then $v_x = v_y$.

Proof: Let z be any event of the server that is a precedent of y or $z = y$. We show that $v_x = v_z$. From the definition of independent, either $z = x$, $z \prec x$ or $x \prec z$. Since z is an event of the server, it can not precede the server-call x. So, $z = x$ or $x \prec z$.

A *chain* to z in the trace is a sequence $w_0 \prec w_1 \prec \cdots \prec w_n$, where $x = w_0$ and $z = w_n$. The length of the chain is n, the number of events excluding the start event. Define *depth* of z to be the length of the longest chain to z in the given trace. We show that $v_x = v_z$ by induction on the depth of z.

For $depth = 0$: We have $x = z$. Hence $v_x = v_z$.

For $depth = n$, where $n > 0$: Every immediate precedent of z has depth smaller than n. Inductively, every immediate precedent has time-stamp v_x. According to action (S), $v_x = v_z$. Setting $z = y$, $v_x = v_y$. □

Lemma 5 Let p be an independent client step. Then, the time-stamping algorithm never changes ld_p once it has been set.

Proof: If p is not a communication step, ld_p is never changed. Let p be a communication step; ld_p can possibly be changed in action (C2) of the algorithm. Let p' be the corresponding server step; since p is independent, so is p', from Lemma 3. Let x be the start event of p (client-call), x' the start event of p' (server-call) and y' the end event of p' (server-response).

$ld_p \geq v_x$	, from action (C1)
$v_{x'} = v_x$	, from action (S), and that x is the only immediate precedent of x'
$v_{y'} = v_{x'}$	, from Lemma 4
$t = v_{y'}$ in action (C2)	, the meaning of t
$ld_p \geq t$ in action (C2)	, from above four propositions
$\max(ld_p, t) = ld_p$ in action (C2)	, from above

□

Lemma 6 For a pending independent step p, $ld_p \leq ud_p$ is invariant in the time-stamping algorithm.

Proof: Both ld_p and ud_p are set in action (C1) by the assignment:

$$ld_p, ud_p := vt + lb_p, vt + ub_p$$

Since $lb_p \leq ub_p$, we then have $ld_p \leq ud_p$. Since p is independent, from Lemma 5, page 17, ld_p does not change. And ud_p does not change in the algorithm. Therefore, $ld_p \leq ud_p$ is invariant for any pending independent step p. □

Theorem 6 (Independence Theorem) Suppose that at any point in an execution there is a pending independent step p such that $ld_p \leq \mu$ for that client. Then the time-stamping algorithm extends the execution.

Proof: Let T be the timed trace corresponding to a network execution when $ld_p \leq \mu$ holds. The proof is by contradiction. Suppose that none of the actions of the time-stamping algorithm that extend the execution, i.e., (S), (C1) and (C3), can be executed. Action (C2) may be executed, without extending execution.

From Lemma 5, page 17, ld_p never changes. A client's μ can change only by executing (C1) or (C3). So, μ does not change, and, hence, $ld_p \leq \mu$ continues to hold. Since (C1) is not executed, there is no start event that can be executed,

i.e., T is quiescent and remains quiescent. Since p is independent, its execution is guaranteed to terminate, so it is passive or will become passive eventually. Therefore, there is a quiescent client for which $ld_p \leq \mu$ eventually holds for a passive step. Hence, (C3) can be executed, a contradiction. □

Corollary 1 Suppose an execution has a pending independent step whose lower bound is 0. Then the execution will be extended.

Proof: Let p be a pending independent step whose lower bound, lb_p, is 0. When p starts, in action (C1) ld_p is set to $vt + lb_p$, that is $ld_p = vt$. From Lemma 5, page 17, ld_p never changes. From Proposition 3, page 12, vt is monotonic. Therefore, $ld_p \leq vt$ is invariant. From Proposition 1, page 11, $vt \leq \mu$ is invariant. Therefore, $ld_p \leq \mu$ is invariant. Apply Independence Theorem. □

Corollary 2 Suppose every pending dependent step has upper bound of ∞. If there is a pending independent step, the execution will be extended.

Proof: Every pending dependent step, q, has upper bound of ∞, so, $ud_q = \infty$. Since there is a pending independent step, from the definition of μ, $\mu = ud_p$, for some independent step. From Lemma 6, page 17, $ld_p \leq ud_p = \mu$. Apply Independence Theorem. □

Corollary 3 Suppose there is a pending step and every pending step is independent. Then the execution will be extended.

Proof: Let p be a pending step such that $\mu = ud_p$. Since p is independent, from Lemma 6, page 17, $ld_p \leq ud_p = \mu$. Apply Independence Theorem. □

4.4 Optimizations

We consider several optimizations of the time-stamping algorithm for special cases. The time-stamping algorithm may be by-passed for steps that have lower bound 0 and upper bound ∞, which is the case most of the time; see Section 4.4.1. In section 4.4.2, we give conditions for time-stamping of end events even when the execution is not quiescent. In Sections 4.4.3 and 4.4.4, we specialize the algorithm for causal and simulation models. We consider mingling real time and virtual time-outs in Section 4.4.5.

4.4.1 By-passing the time-stamping algorithm

The time-stamping algorithm requires consideration of the lower and upper bounds of every step. In a vast majority of cases, nearly all steps have lower bound 0 and upper bound of ∞. That is, the time-stamp assigned to its events are irrelevant. In such cases, the step does not affect the time-stamps of any other event (because it does not affect μ), so, it may be removed from consideration in the algorithm. Its start event is time-stamped vt, as usual, and, from Section 4.4.2, its end event may be time-stamped with vt whenever it becomes passive.

4.4.2 Time-stamping end events in non-quiescent states

The time-stamping algorithm time-stamps an end event in action (C3) only, when the client is quiescent. The end event of a passive step p may be assigned time-stamp vt at any point in the execution provided $ld_p \leq vt$. In particular, this optimization allows us to time-stamp the end event as soon as a step is completed provided its lower bound is 0. So, $Vwait(0)$ can be implemented as a skip using this rule. It can be shown that all the safety and progress conditions are met by this rule. This time-stamping does not change vt.

4.4.3 Causal Model

We have described the causal model in Section 1.1.1, page 2. Causal model obeys only the causality and monotonicity conditions of Section 3, page 8. There is no $Vwait$, and consequently there is no eagerness or duration requirement. The exact values of the time stamps are irrelevant except that an event should have a strictly lower time-stamp than any event it precedes. We meet these requirements by having every step have a lower bound of 1 and upper bound of ∞. (If lower-bounds are allowed to be 0, then the safety conditions can be trivially satisfied by setting all time-stamps to 0.)

Since the exact values of the time-stamps are irrelevant, we may advance virtual time at any point, i.e., there is no need to wait till quiescence to increase vt. There is no need to maintain lower and upper-deadlines for individual steps either. We may advance vt as soon as a step starts. Further, we need not distinguish between active and passive steps. We get the following algorithm.

Server

(S) For any server event x, set $v_x = \max\{v_y |\ y \prec x\} + 1$.

Client

(C1) Executing start event of step p:
Set $vt := vt + 1$; $v_x := vt$, where x is the start event of p.

(C2) Executing end event of step p:
If p is a communication step and t the time-stamp of the corresponding server-response,
then set $vt := \max(vt, t) + 1$ else set $vt := vt + 1$;
set $v_y := vt$, where y is the end event of p.

The model and the algorithm are (almost) identical to those proposed by Lamport [9]. The only servers in that paper are unbounded asynchronous channels for message communications[4]. Our algorithm applies more generally for arbitrary servers; therefore the server step (S) is more elaborate in our case. For message communication, (S) may be simplified to: for receive event y at the channel corresponding to send event x, set $v_y := v_x + 1$.

[4]The "put" and "get" steps are "send" and "receive" in [9].

4.4.4 Simulation Model

In the typical model for discrete event simulation, there is no server and there is just a single client. Therefore, there is no communication step. Every step except *Vwait* consumes 0 virtual time. There is no need for the causality condition because, with a single client, monotonicity condition subsumes it.

There is no need to maintain the lower and upper-deadlines for individual steps; instead we maintain completion-time, ct, for every *Vwait* step. The passive steps are all *Vwait*s; any other step is active and its end event is time-stamped as soon as it is completed, according to Section 4.4.2, page 19. The simplified time-stamping algorithm is same as the traditional single-processor discrete event simulation algorithm.

Client

(C1) Starting step p:
Designate p passive if p is *Vwait*, otherwise it is *active*.
Set $v_x := vt$, where x is the start event of p.
Set $ct_p := vt + k$, if p is $Vwait(k)$.

(C2) Executing end event of active step p:
Set $v_y := vt$, where y is the end event of p.

(C3) At quiescence, if there is a pending step:
Let ct_p be the minimum over all ct_q, where q is passive.
Set $vt := ct_p$; $v_y := vt$, where y is the end event of p.

4.4.5 Real time computations

Computations involving real-time use time-out using *Rwait*, analogous to *Vwait*. Consider the case where there is a single real-time clock at every client, but the real-time clocks at different clients are not necessarily synchronized.

Suppose events e_1 and e_2 occur at a client at real times r_1 and r_2 and have (virtual) time-stamps v_1 and v_2. From monotonicity, $r_1 < r_2 \Rightarrow v_1 \leq v_2$. Taking the contrapositive (and switching the roles of v_1 and v_2), $v_1 < v_2 \Rightarrow r_1 \leq r_2$. Note that events occurring at the same real time may have different time-stamps, and events occurring at different real times may have the same time-stamp.

To preserve the semantics of real-time time-out, computation must resume immediately following a *Rwait*. That is, it should be possible to time-stamp the end event of any pending *Rwait* as soon as it occurs. In order to apply the optimization of Section 4.4.2, page 19, we require that $ld_p \leq vt$, for any *Rwait* step p. This is possible only if the lower-bound for every *Rwait* is 0.

Further, we do not expect the presence of a real-time time-out in a client program to restrict execution of any other step of that client. That is, *Rwait* should not have an upper-deadline such that a competing step in a concurrent thread is prevented from continuing in action (C3). Therefore, we take all *Rwait*s to have upper-bound of ∞.

Given these time bounds, execution of *Rwait*s may bypass the execution of the time-stamping algorithm, as we have described in Section 4.4.1, page 18.

Example 3 Consider two concurrent threads in a client. One thread executes *Rwait*(2) and the other *Vwait*(5) followed by *Rwait*(1). Let $r1$, $r2$ and $v5$ denote the start events of *Rwait*(1), *Rwait*(2) and *Vwait*(5) respectively, and the corresponding primed variables denote their end events. We show the possible timed traces in Table 2; events at the top, time-stamps at bottom. Events $r2$ and $v5$ may be transposed in the last four traces.

$r2$	$r2'$	$v5$	$v5'$	$r1$	$r1'$
0	0	0	5	5	5
$r2$	$v5$	$r2'$	$v5'$	$r1$	$r1'$
0	0	0	5	5	5
$r2$	$v5$	$v5'$	$r2'$	$r1$	$r1'$
0	0	5	5	5	5
$r2$	$v5$	$v5'$	$r1$	$r2'$	$r1'$
0	0	5	5	5	5
$r2$	$v5$	$v5'$	$r1$	$r1'$	$r2'$
0	0	5	5	5	5

Table 2: Intermingled Real time and Virtual time time-outs

5 Applications

The monotonicity condition allows us to impose order on causally unrelated events at a client using explicit values of virtual time; Section 5.1 shows an example. We consider the problem of distributing a simulation among several clients in Section 5.3. A longer combinatorial example, computing shortest path in a graph, is taken up in Section 6.

5.1 Ordering causally unrelated events

We give a small example in which use of virtual time simplifies program design. A set of concurrent threads, numbered from 0 through n, are to be executed. It is required to start the threads sequentially, thread $i+1$ after thread i, $0 \leq i < n$. Since the threads are independent, there is no causal order among them. If each thread is executed on a separate client, a causal order among their start events has to be imposed. One possibility is to have thread i send a token to thread $i+1$ after it starts, $0 \leq i < n$, and thread $i+1$ starts only after receiving the token. If all threads are executed on a single client, a simpler strategy is applicable. Thread i waits for i units of time (real or virtual) before starting. Waiting for virtual time has the advantage over waiting for real time in that (1) no real time is wasted, and (2) sequencing is guaranteed by the conditions

imposed on virtual time, whereas threads may start out-of-order if the unit of real time is too small and the time-out mechanism is inexact.

We show that this scheme meets the specification. From the eagerness condition, all $Vwait(i)$, $0 \leq i \leq n$, start at virtual time 0. If $Vwait(i)$ completes, from the duration condition it completes at virtual time i. From the eagerness condition, thread i starts at virtual time i. From monotonicity condition, all events at virtual time i come before those at virtual time $i+1$ since the threads are implemented on one client; therefore, thread $i+1$ starts after thread i. Additionally, if it is specified that all steps in each thread consume 0 virtual time, then all events in thread i come before any event of thread $i+1$. That is, the threads are executed sequentially.

If the threads are implemented on two clients, say threads 0 through $k-1$ on client 1 and k through n on client 2, we can combine both strategies: thread i in client 1, $0 \leq i < k$, waits for i units of virtual time and then starts, client 2 starts only after receiving a token from (thread $k-1$ in) client 1, and then thread j, $k \leq j \leq n$, waits for $j-k$ units of virtual time before starting.

5.2 Plane-sweep algorithms in Computational Geometry

Plane-sweep is a powerful algorithmic technique in computational geometry. Algorithms using plane-sweep can always be described using virtual time, where the current value of virtual time denotes the position of the line that sweeps the plane. The advantage of such a description is that book-keeping aspects of plane-sweep can be completely ignored. We study an algorithm, due to Bentley and Ottman [2, 14], for enumerating the intersection points of a given set of line segments in a plane.

Line Segment Intersections We are given a finite set of line segments in a plane, where each segment is described by the pair of its end points. It is required to enumerate the points of intersection of all pairs of segments. The algorithm has the following salient features.

1. It maintains a list data structure, A. The list is initially empty.
2. All end points and intersection points are processed in order of their x-coordinates.
3. Processing a point may update A and create new intersection points.

The algorithm terminates, because there are a finite number of end points and intersection points, and each processing step terminates. The exact details of processing are not germane for our discussion; it can be shown that all intersection points are generated by the processing steps.

For sequential processing of the points in order of their x-coordinates, all end points are initially stored in a queue. At any step, the point with the smallest x-coordinate is removed from the queue and processed, which may cause intersection points to be added to the queue. This is reminiscent of sequential

simulation. We can simplify the description by creating a thread for each point $(x,\ y)$ which is processed at virtual time x; assume that x-coordinates of all such points are distinct. All threads are executed at a single client, and computation steps take no virtual time. A sketch of the algorithm is given below, where c denotes the current virtual time.

Initially, $c = 0$ and A is the empty list;
    concurrently, for each end point p: $process(p)$.

Procedure $process(p)$ =
    $Vwait(t - c)$, where t is the x-coordinate of p; $c := t$;
    update A, and using A compute a set of new intersection points S;
    concurrently, for each q in S: $process(q)$.

5.3 Distributed Simulation

Distributed simulations [4, 11, 3, 7] have become essential for analyzing systems consisting of thousands, or even millions, of components. A distributed simulation is implemented on several processors, each implementing a client. Typically, the clients communicate directly via asynchronous channels, without using explicit servers. Each client is an event[5] processing engine: it processes an event, possibly creating new events to be processed by other clients at specific virtual times, and then communicates these events and times to other clients.

An event happening at one client has to be communicated to another client with its virtual time of occurrence. A client can not advance its virtual clock unless it is assured that it will receive no future communication in real time from another client at a lower virtual time. In terms of our model, a *get* step on a channel can not specify its upper bound; the step is completed at exactly the time dictated by the corresponding *put* step. So, a dependent step, *get*, does not have ∞ as its upper bound, the independence theorem of Section 6 is not applicable, and extension of execution can not be guaranteed. In fact, deadlock is the major problem with distributed simulation; different heuristics are employed to overcome deadlock.

The algorithm proposed in this paper can be applied in a limited sense to distributed simulation. A simulation problem has to be partitioned among clients where communications across clients are periodic so that no client has to wait too long to acquire the virtual clock value of any client with whom it communicates.

6 Shortest path

We show a solution to the well-known shortest path problem expressed as a concurrent algorithm using virtual time. The algorithm of Dijkstra [5] is in fact a sequential implementation of this algorithm.

[5]Here, "event" refers to real-world events, not the events in our formal model.

It is required to find a shortest path from a *source* node to a *sink* node in a finite directed graph in which each edge has a positive length. We will merely record the length of the shortest path from source to every node reachable from it; determining the shortest path itself is a small extension of this scheme.

Imagine that the length of an edge is the amount of time taken by a light ray to traverse that edge. First, the source records 0, the length of the shortest path to itself. Then it sends rays along all its outgoing edges simultaneously at time 0. On receiving a light ray along any of its incoming edges, a node records the time and sends rays along all its outgoing edges immediately. It can be shown that every path length to a node is eventually recorded by the node in order of (non-decreasing) length. The computation may be terminated as soon as the sink node records its first value, which has to be the shortest path length to it. A node may stop its computation after recording its first value, because only its shortest path may be included in the shortest path to any other node.

6.1 The Shortest Path Program

We implement the proposed scheme below, where $eval(u)$ is a procedure.

```
def eval(u) =
  record current virtual time value in a FIFO channel for u;
  in parallel, for every successor v of u with edge length d:
    wait for d time units and then call eval(v)

Initially: eval(source)
```

Waiting for d units of time is accomplished by $Vwait(d)$. Other steps, including for recording, consume no virtual time. The algorithm can be described in any programming language that supports concurrency, virtual time and virtual-timeout. Appendix B has a (concise) program in programming language Orc [12].

Let $ch(v)$ be the channel corresponding to node v. If the algorithm is implemented on a single client, but possibly multiple servers corresponding to the channels, it records the lengths of paths to any node v in $ch(v)$ in order of their magnitude. The first value in $ch(sink)$ is the length of the shortest path to sink. If there is no path from source to sink, no value will ever be stored in $ch(sink)$.

Simulating the concurrent algorithm on a single processor gives us Dijkstra's algorithm [5]. The concurrent algorithm is simple to develop and justify. The simulation merely involves additional book-keeping that is best left to the time-stamping algorithm.

6.2 Correctness of Shortest Path Program

We prove that the proposed program, using the time-stamping algorithm of Section 4.1, eventually computes the length of the shortest path to sink if there is a path from source to sink. We assume that any recording step has both lower and upper bounds of 0. Note that all steps are independent.

Lemma 7 Every pending *Vwait* is eventually completed.

Proof: The program has no dependent step. Therefore, from Corollary 2, page 18, the execution will be extended if there is a pending step. There is at most one recording step per node that can be completed before some *Vwait* is executed. Therefore, only a finite number of extensions by recording steps are possible before all steps become pending. Some pending $Vwait(k)$, where $k > 0$, is eventually completed, thus increasing vt. The value of vt can not increase past the upper deadline of any pending *Vwait* step, from the duration condition of Section 3, page 8. Therefore, every pending *Vwait* will be completed. □

Theorem 7 Let $dist(v) = \{t|\ t$ is a path length to $v\}$. Then,
$t \in dist(v)\ \equiv\ (eval(v)$ is called at virtual time $t)$.

Proof: Proof is by induction on t.

• Base case, $t = 0$: First, we show that $0 \in dist(v)\ \equiv\ eval(v)$ is called at 0. Since every edge length is positive, $0 \in dist(v)\ \equiv\ v = source$. So, we have to show that $eval(source)$ is called at time 0 and no other node is called at time 0. The first part follows from the initial condition of the program. Next, any other call to *eval* is made from *eval* after a positive delay, because for every $Vwait(k)$, $k > 0$. Therefore, the only call made at time 0 to *eval* is made initially.

• Inductive case, $t > 0$:

$$
\begin{array}{ll}
 & t \in dist(v) \\
\equiv & \{\text{definition of } dist(v)\} \\
 & t \text{ is a path length to } v \\
\equiv & \{t > 0 \text{ means the path has at least one edge.} \\
 & \quad \text{Let the penultimate node in the path be } u, \text{ and} \\
 & \quad \text{let } t' \text{ be the length of the path to } u \text{ along this path}\} \\
 & \text{there exist } u \text{ and } t', \text{ where } t' \in dist(u) \text{ and } t = t' + d(u, v) \\
\equiv & \{\text{all edge lengths are positive, so } t' < t\text{; induction hypothesis}\} \\
 & \text{there exist } u \text{ and } t', \text{ where } eval(u) \text{ is called at } t' \text{ and } t = t' + d(u, v) \\
\equiv & \{\text{see below}\} \\
 & eval(v) \text{ is called at } t
\end{array}
$$

The proof of the last step is by mutual implication. If there exist u and t' where $eval(u)$ is called at t' and $t = t' + d(u, v)$ then, using eagerness, $eval(v)$ is called at t. Conversely, if $eval(v)$ is called at t, from the program there exist u and t' such that $eval(u)$ is called at t' and $t = t' + d(u, v)$ □

This theorem has established that all elements of $dist(v)$, for any v, will eventually be added to $ch(v)$. But it has not established that they will be added in order. We need to execute the program at a single client so that we can establish this result using the monotonicity condition.

Theorem 8 Given that the program is executed at a single client, every channel contains items in order.

Proof: An item is added to channel $ch(v)$ only through $eval(v)$. From Theorem 7, if item t is added to $ch(v)$ it is added at virtual time t. Further, the recording step on $ch(v)$ consumes 0 virtual time. So, item t is stored in the channel by virtual time t. From the monotonicity condition, Section 3 (page 8), if the program is executed at a single client, items are stored in order of virtual times, and, hence, in order of their values. □

From Theorems 7 and 8, the length of the shortest path from source to sink, if there is one, is eventually added to $ch(sink)$ as its first item.

A Appendix: Proof of Lemma 1

Lemma 1: Network traces satisfy the conditions on traces, i.e.,

1. Tx is a network trace implies T is a network trace, and for all z, where $z \prec x$, $z \in T$.

2. Txy is a network trace and Ty is not a network trace implies $x \prec y$.

Proof of (1): Given that Tx is a network trace, from the definition of network trace, we first conclude that $(Tx)_u$ is a trace of u, for every entity u. Now, T_u is a prefix of $(Tx)_u$. From prefix closure for u, T_u is a trace of u. Next, Tx is a network trace; so, for any event in Tx all its precedents come before it in Tx. Therefore, for any event in T all its precedents come before it in T. Combining these two results, T is a network trace and for all z, where $z \prec x$, $z \in T$.

Proof of (2): We have to show:

$(Ty)_u$ is a trace of u for every entity u, and (2.1)

For any event in Ty all its precedents come before it in Ty. (2.2)

Proof of (2.1): Let y be an event of entity w. First, we prove that $(Ty)_u$ is a trace of u for every entity u, $u \neq w$. Since Txy is a network trace, its prefix T is a network trace, from Proof of (1). So, T_u is a trace of u. And, $(Ty)_u = T_u$.

Next, we show that $(Ty)_w = T_w y$ is a trace of w. Consider two cases.

Case 1) x is an event of w: Since Txy is a network trace $(Txy)_w = T_w xy$ is a trace of w. We have $x \not\prec y$ in the network; hence $x \not\prec y$ in entity w because both x and y are events of w. From the trace condition in w, $T_w y$ is a trace of w.

Case 2) x is not an event of w: Since Txy is a network trace, $(Txy)_w = T_w y$ is trace of w.

Proof of (2.2): Given that Txy is a network trace, for every event in T all its precedents come before it in T. For y, given that $x \not\prec y$, all precedents of y are in T. Therefore, for any event in Ty all its precedents come before it in Ty.

B Appendix: Shortest Path Program in Orc

We describe the shortest path program in programming language Orc [12, 8]. Below, we use `u` for the channel corresponding to node `u` that is used for recording values of shortest paths to node `u`. Step `u.put(t)` appends `t` to channel `u`. Step `Vtime()` returns the current value of virtual time. It completes in all cases and consumes no virtual time, i.e., its associated lower and upper bounds for virtual time consumption are both zero. The symbol `>>` denotes sequencing. Calls to `Succ(u)` returns all pairs `(d,v)`, where `v` is a successor of node `u` and `d` is the length of the edge from node `u` to node `v`. For each returned pair a new thread consisting of `Vwait(d) >> eval(v)` is initiated. Execution of `Succ(u)` consumes no virtual time.

```
def eval(u) =
  u.put(Vtime()) >>
  Succ(u) >(d,v)>
   Vwait(d) >> eval(v)

eval(source)
```

Acknowledgement This paper is the result of intensive discussions with Albert Benveniste, William Cook, Claude Jard, David Kitchin, Sidney Rosario and John Thywissen. I am particularly grateful to Adrian Quark who suggested the first model of virtual time in Orc and implemented it. My earlier work with Mani Chandy on distributed simulation has helped immensely in my understanding of both virtual time, and, more broadly, computer science.

References

[1] Paulo Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks: A logical clock for dynamic systems. In *Principles of Distributed Systems*, number 5401 in Lecture Notes in Computer Science, pages 259–274. Springer-Verlag, 2008.

[2] J. L. Bentley and T. A. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Trans. on Computers*, 28:643–647, 1979.

[3] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical Report TR-188, MIT, LCS, Cambridge, Mass., 1977.

[4] K. Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE*, SE-5(5):440–452, Sep 1979.

[5] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:83–89, 1959.

[6] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In K. Raymond, editor, *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, page 5666, February 1988.

[7] D. R. Jefferson. Virtual time. *Prog. Lang. Syst.*, 7(3):404–425, 1985.

[8] David Kitchin, Adrian Quark, and Jayadev Misra. Quicksort: Combining concurrency, recursion, and mutable data structures. In A. W. Roscoe, Cliff B. Jones, and Ken Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing. Springer, 2010. Written in honor of Sir Tony Hoare's 75th birthday.

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

[10] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France*, page 215226. Elsevier, October 1988.

[11] Jayadev Misra. Distributed discrete event simulation. *Computing Surveys*, 18(1):39–65, Mar. 1986.

[12] Jayadev Misra and William Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling (SoSyM)*, 6(1):83–110, March 2007.

[13] Carroll Morgan. Global and logical time in distributed algorithms. *Information Processing Letters*, 20(4):189–194, May 1985.

[14] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[15] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: Constant size logical clocks for distributed systems, 1996.