A Deterministic Model of Time for Distributed Systems

Jeremy Carroll, Andrei Borshchev*
Telecom Management Department
HP Laboratories Bristol
HPL-96-112
July, 1996

time, logical time,
real time,
distributed systems,
causality,
Isis, middleware,
delayed delivery

This paper proposes a linear, deterministic, logical time model for distributed systems. We give an account of causality within distributed systems which undergirds the time model. We discuss some advantages for the application programmer in using our time model.

Internal Accession Date Only
*St. Petersburg Technical University, Russia
Presented at the *8th IEEE Symposium on Parallel and Distributed Processing*, October 1996.

A Deterministic Model of Time for Distributed Systems

Jeremy J. Carroll
Hewlett-Packard Labs, Bristol, U.K.
jjc@hpl.hp.com

Andrei V. Borshchev
St.Petersburg Technical University, Russia
andrei@trantor.nord.nw.ru

Abstract

This paper proposes a linear, deterministic, logical time model for distributed systems. We give an account of causality within distributed systems which undergirds the time model. We discuss some advantages for the application programmer in using our time model.

1. Introduction

We propose a new deterministic, logical time model for distributed systems.

The time model is defined at the level of temporal semantics in terms of message ordering.

These temporal semantics allow the application programmer to think about time linearly without having to worry about concurrency or race conditions. This makes the temporal semantics of distribution much closer to those of non-distributed systems.

These semantics are to be implemented in middleware which offers a messaging facility. For the application programmer the messaging appears to obey the linear time order of our time model. In reality the messaging operates with a high level of concurrency. The cost of the time model is mainly felt in latency, not throughput.

The time model depends upon a new definition of causality which differs significantly from Lamport's causality [11].

2. Motivations

We look at two distinct motivations for this work, one theoretical, the other practical.

2.1 Causality

Lamport, in his landmark paper, [11], defines his 'happened before' relationship and notes 'that it is possible for event *a* to causally affect event *b*'. This led, via Fidge [7] and Mattern [12] to the misnamed 'causal' ordering of ISIS [2,13]. The lack of true causality in ISIS is shown by Cheriton and Skeen [6], '*it can't say the "whole story"*'. We modify their Figure 4 as our Figure 1.

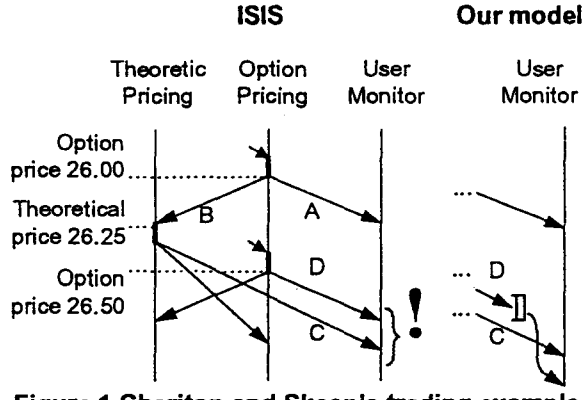

Figure 1 Cheriton and Skeen's trading example

Their criticism is that the application semantics mean that message C is caused by message B, whereas the User Monitor sees message D, that supersedes message A, before message C. Moreover, 'causal ordering' accepts this; for Lamport message C and message D are concurrent.

We see that the correct notion of cause notes that C happens *because of* B and D merely happens *after* B. This reminds us of Hume's second definition of causation [9]:

> *We may define a cause to be [...] if the first object had not been, the second never had existed.*

Under our time model we are required to deliver message C, which is strongly caused by B (in this Humean sense), prior to message D, sent after B, (see the axiom of strong causality in section 4). In the figure this is shown by delaying message D, using an ISIS-like mechanism. (We require that the multicast A&B is made in the order A then B).

2.2 Telecom billing systems

The authors are part of the HP Labs team working in the area of telecom billing systems (see Beech [1]).

Almost all telecom billing systems are currently implemented on centralised mainframes. The mainframe receives its input data as Call Detail Records (CDRs) from the telecom switching network. The telecom switches output 'raw' CDRs; these are transmitted unprocessed to the central mainframe, where all processing is done. The typical data transmission system consists of motorbike couriers with magnetic tapes (a high bandwidth, high latency solution).

The billing architecture proposed by the HP lab team involves using geographically distributed processing, in which as much processing as possible is done near the data sources. The processed CDRs are then consolidated centrally. (The motorbike network is modernised!)

Ideally the same application programmers should continue to develop the billing systems as they migrate from a centralised architecture to a distributed one

Hence any serious proposal for an application programmers' environment should assume that the application programmer is happier with COBOL than with concurrency, and happier with records than race conditions.

This time model is proposed with such a target user in mind.

3. Background assumptions

We take an application or system level view of the distributed system, rather than only seeing a set of processes. This is reflected in the time-line being a property of the whole system. This global view is however, a development and system management view of the system, and not a statement about implementation. We prefer decentralised implementations in which global properties emerge out of localised processing, rather than from a central server or master/slave arrangement.

We consider distributed systems which are sets of processes connected by FIFO channels. Some of the processes may have external channels through which they communicate with the system's environment. Individual processes are driven by input messages, and the whole system is driven by the external input messages. While handling a message, a process can send an arbitrary number of output messages, which are considered as consequences of the input message.

3.1 Process assumptions

A process consists of a number of application layer event handlers with shared state.

The only events we consider are message related events. We use the term *message arrival* to indicate the message arriving in the presentation layer, and *message delivery* to indicate the message being passed from the presentation layer to the application layer. The application layer may generate *message send request* events which are passed to the presentation layer, which in turn generates *message send* events. The application layer also generates *message processing complete* events, which are passed to the presentation layer.

The presentation layer delivers messages, one by one, to the application layer by means of message delivery events. After the application layer receives a single message delivery event, it:

- generates zero or more message send requests.
- changes the process state.
- generates a message processing complete event that indicates that it has finished processing that message and is ready to receive another message. (This is required to happen in bounded time).

Such message handler *invocations* are the atomic units out of which we see a distributed system being formed. A process history is simply a sequence of such invocations, each of which may affect subsequent invocations by changing the internal state of the process.

3.2 Channel assumptions

A channel links the presentation layers of two processes.

We assume that the channels are perfect FIFO channels. A message send event at one end is eventually followed by a message arrival event at the other.

At the application layer the channels are simplex; however auxiliary messages, from one presentation layer to the other, are allowed in both directions.

3.3 Clock assumptions

The single assumption of a global nature is that of the existence of a global real-time clock accessible from anywhere within the system.

We only require that the clock is locally monotonic increasing, and that there is some bound (perhaps only statistical) on the difference between two simultaneous readings of the clock in different processes. Neither the time model, nor the implementation we present, make any restrictions as to the size of the bound on the error. However, many applications, including telecom billing, may make more severe demands on the error bounds.

3.4 Typical applications

While these assumptions are general, the emphasis on simplex channels contrasts with the client/server

paradigm. This is because our applications (distributed test environments; event correlation services [8]; environmental monitoring systems as well as billing systems) gather and process data from a number of geographically separate sources. Such an application is illustrated in Figure 2. These applications tend to have few loops in their geographic dataflow diagrams. They are also tolerant of latency, which is reflected in a significant amount of queuing and delayed delivery within the implementation of the time model.

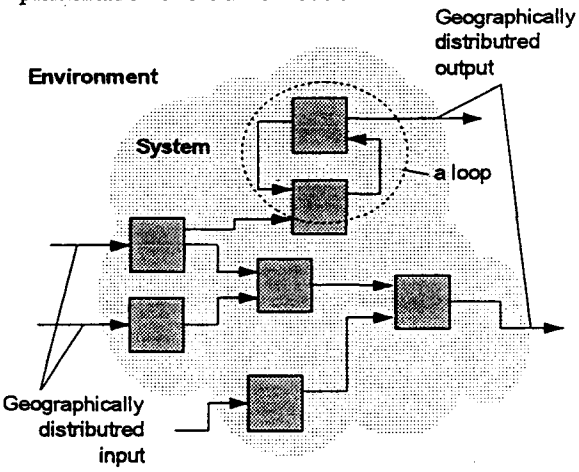

Figure 2 A typical data analysis application

4. The time model

The time model is a strict total ordering of the set of messages in the system (we omit the word 'strict' in the following).

If messages are delivered to every process in time order and messages are sent along every channel in time order then the system is said to obey the time model.

A total ordering on the set of messages is equivalent to an injective mapping from the set of messages in the system to a totally ordered set (the time-line).

This gives the first principle of the time model: *there is a unique time for everything*.

Such a time is simply a label for the message and does not have any necessary relationship with real time. Such a label can be useful for evaluating the time order relationship between messages.

We now introduce the total order relation "$<$" on the set of all messages in the system. This relation is based on two partial order relations, "$\Rightarrow$" (sent before) and "$\rightarrow$" (strong causality).

"$\Rightarrow$" is defined by:

- For any two external input messages, $m0$ and $m1$, either $m0 \Rightarrow m1$ or $m1 \Rightarrow m0$. This is given by the environment, typically by the clock time of message arrival. In other words, external input messages are totally ordered with respect to $\Rightarrow$.[1]
- If message send requests for messages $m0$ and $m1$ occur in that order during a single invocation of the message handler of some third message, then $m0 \Rightarrow m1$.

"$\rightarrow$" is the least partial order such that:

- If the message send request for $m1$ occurs during the invocation in response to $m0$, then $m0 \rightarrow m1$. (i.e. a message strongly causes any messages sent by its handler).

"$<$" is the total order such that:

1. If $m_0 \Rightarrow m_1$ then $m_0 < m_1$.
2. If $m_0 \rightarrow m_1$ then $m_0 < m_1$.
3. If $m_0 \Rightarrow m_1$, $m_0 \rightarrow m'_0$ then $m'_0 < m_1$. *(The strong causality axiom)*

The first two axioms correspond to Lamport's axioms; the third is the heart of the time model.

The idea behind the strong causality axiom is the following: if a process or the system's environment sends two messages, one after another, then any consequence of the first message should happen before the second message and any of its consequences. We will see that this property is highly desirable and intuitive.

This gives the second principle of the time model: *there is enough time for everything* (i.e. enough time for all remote consequences to happen before the next local event).

The totally ordered message set can be represented as a tree, see Figure 3. The root of the tree is the system's environment which generates external messages. The nodes are invocations of message handlers. Arcs represent messages.

Using this tree it is easy to reconstruct the message relations. For example, $a \rightarrow b$ because b was sent while a was handled; $b \Rightarrow c$ because b was sent before c in the same invocation. Also, $a \rightarrow f$ and $x \Rightarrow z$ as these relations are transitive. f and e are incomparable under both "$\Rightarrow$" and "$\rightarrow$"; nevertheless $f < e$. To compare two messages with respect to the global order relation "$<$" one has to trace paths

[1] More exactly if m_0 and m_1, arrive from the environment at the same process in that chronological order (by clock time) then $m_0 \Rightarrow m_1$. This generates a unique least partial order. There is a choice of total orders respecting this constraint; we choose one such total order. The choice is arbitrary, but some choices might give better results than others. For the specific applications we have in mind, having adequately synchronized real time clocks and developing a total order that respects these clocks is best.

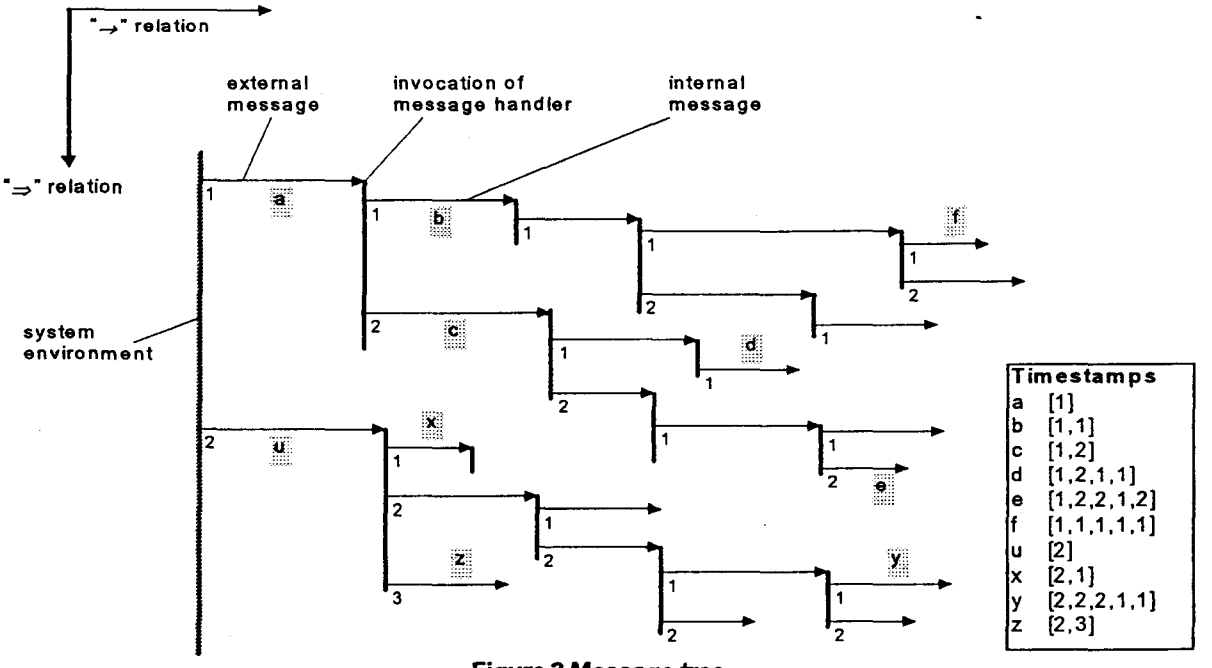

Figure 3 Message tree

from the root to these messages. There can be three possible cases, which correspond to the three axioms. They are shown by the following three examples taken from Figure 3:

- c lies on the path of d, hence, $c \rightarrow d$ and therefore $c < d$.
- x and z have the same path, but x is sent before z, so $x \Rightarrow z$ and $x < z$.
- f and e have the same path prefix, but then their paths fork, and b (with $b \rightarrow f$) is sent before c (with $c \rightarrow e$), which means $b \Rightarrow c$, so $f < c < e$, giving $f < e$.

If a distributed system follows this time model, i.e. if messages are delivered to each process in this order, and sent down each channel in this order, then the system's behaviour will be deterministic, independent of the speed of processes and channels.

In the implementation discussed below, we use the set of sequences of integers as a time-line. Examples are shown in Figure 3. The sequence is determined by the path from the root to a message. This is ordered using the standard alphabetic ordering on integer sequences.

5. Implementation

We sketch one possible implementation of the time model. It is pessimistic, i.e. based on the delayed delivery of messages. An optimistic implementation could be achieved using Jefferson's virtual time with rollback [10], (in which case the assumptions about channels, section 3.2, are unnecessary).

The timeline used is the set of integer sequences. However since we cannot number the (distributed) external messages sequentially we assign a unique pair to each, the value of a real clock and an external input identifier.

Each node in the distributed system is structured as shown in Figure 4. All functionality related to support for the time model resides in the time service. The time service has a local logical clock which is updated whenever a message is received or sent by its process. The "timestamp assigner" at the border with the system's environment has a real clock and a unique input identifier.

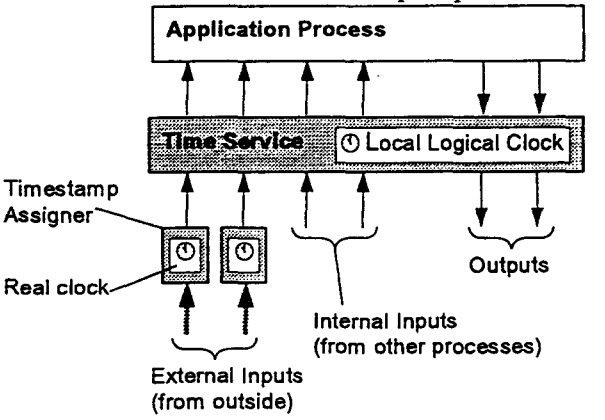

Figure 4 A process with its time service

As in the ISIS ABCAST protocol [13], input messages are not delivered to the process until there are messages present on all inputs. Once this condition holds, the local

clock is set to the timestamp of the oldest message, a new integer is added to the sequence, and the oldest message is delivered. When the process sends a message it is timestamped by the time service with the current value of the local clock; and then the clock is incremented (i.e. the last integer in the sequence is incremented).

This algorithm fully implements our time model, but does not work when the system contains loops. Moreover, rare messages either on an external input or on an internal link may slow the system down. To solve these problems we have developed a specific channel flushing algorithm for our time model, significantly different to the one of ISIS ABCAST.

The general idea is that each time there are input messages waiting while some inputs are empty, the time service sets a flush timer. On the timeout event it sends flush requests to all empty inputs, and then waits for positive or negative responses. If positive responses come from all inputs the oldest message is delivered. If a negative response comes on any input the flush is re-scheduled. Requests from other time services are handled in the following way. First, the time service tries to reply using the local information (its local clock and the timestamps of waiting messages). If it is unable to do so, it forwards the request to all empty inputs. If all responses are positive, so is the one to the remote requester. Otherwise, the response is negative.

6. Performance modelling

We used COVERS simulation environment [3,4] to assess the impact of the time model (implemented as described above) on system performance.

The modelling results show that, while the system is underutilized, the overhead traffic (the auxiliary channel flushing messages sent at the time service layer) can be as much as 8 times the number of the application messages. However, when the input data rate is high and the system reaches its maximum throughput, the time model traffic falls down to an acceptable amount.

Performance modelling also shows that the system's throughput depends on the value of the flush timeout, which should be chosen carefully, especially if the system contains loops. Dynamic (adaptive) tuning of the flush timer depending on the percentage of successful channel flushes may be considered.

7. Total ordering and programming models

The strong causality principle provides a much easier framework for the application programmer. A programmer used to non-distributed systems is used to a time-line that is totally ordered, and arbitrarily divisible. The first part of a procedure starts and completes before the second part. There is no special temporal semantics for a normal part of such a program. For example, a subprocedure invoked by the first part will also have completed before the second part.

When we move to a distributed system, programmers are normally expected to have to understand either concurrency or asynchronicity or both. However, with our time model the programmer's view of the system is just like that for a non-distributed system (Figure 5). In this figure we see that the end of a remote event handler logically returns 'control' to the sender. However, this logical return of control does not necessarily imply any loss of actual concurrency: in the example shown, the only ordering constraint on the events is that the second 'send msg 2' instruction is received after the first. This is ensured by the implementation of the time model.

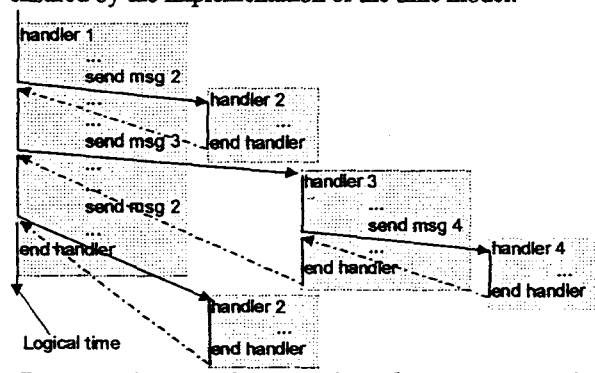

Figure 5 Temporal semantics of message sends

We believe the COBOL programmers of telecom billing systems will find this temporal semantics intuitive and easy to use. Moreover, the global linear time line removes many of the different orderings that plague distributed systems. Much of the logic for dealing with complex ordering constraints has moved from the application layer, where it confuses, to the middleware in the presentation layer, where it empowers. In this way, applications built on top of our time model can be significantly simpler; easier to build, understand and change. This makes them substantially cheaper and less buggy.

This view of message send like procedural call clarifies why our time model solves the problem of Cheriton and Skeen explored in section 2.1.

8. Loops

We have seen in the discussion in section 5 that loops generate issues for the time model. Typical loops generate a need for many auxiliary messages. Most loops only permit the processing of a single message at any one time

anywhere within the loop (we say the loop 'locksteps'). The channel flush algorithm allows the processors to negotiate as to which goes first.

We have three different solutions to these problems: we can remove spurious loops in the design process; or collocate processes in a loop; or explicitly break the link of strong causality between a message and its feedback.

The traditional design models, client/server, master/slave, encourage a control driven view of a distributed system, which leads to loops. A more data driven view of a system, like the data flow diagrams of structured analysis, is typically less loopy.

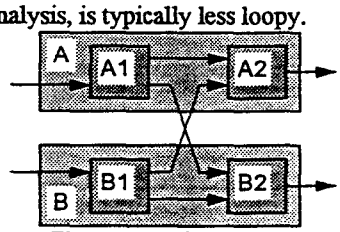

Figure 6 A false loop

Moreover, where a first cut has loops, a more detailed analysis of a distributed system may show that these loops are spurious. For example, in Figure 6, there is an apparent loop between process A and process B, (a flow from A feeds into B which feeds back into A). This vanishes when we look at the subprocesses A1, A2, B1, B2.

We can put all the processes in the loop on the same processor. There will be no additional penalty in terms of loss of parallelism, since the time model enforced this loss. This approach will minimise the cost of the auxiliary messages, because they will now be local messages.

Informally, the problem with a loop is feedback. Feedback happens when an input message to a process strongly causes another message (the feedback) to arrive later at the same process. Under the strong causality axiom, feedback is strongly caused by the original messages, and hence comes before all subsequent messages. Hence any process in a loop must, after processing every message, first ascertain whether there is any feedback, before proceeding to deal with any other input.

In [5] we construct a delay process for which each input message is followed by an identical output message, which is however not 'strongly caused' by the input, but rather scheduled by it to happen later. Hence, with a delay in the loop, a process can know that any feedback will not arrive until after the duration of the delay. Thus it can accept other messages arriving before the feedback.

9. Conclusion

We have presented our time model showing its advantages from the point of view of programming models and causal semantics. We have also sketched our initial implementation.

We discuss the implementation in more detail, giving detailed performance modelling results, and an extended discussion in [5].

We are now working on a second implementation within a CORBA environment, in which we are considering the issue of the request/response 'loop' ubiquitous in client/server architectures.

We are also extending the theoretical work. Our next steps are to look at: deadlock prevention within this time model; and control flows in the opposite direction from the data flows.

References

[1] Mike Beech. *Architecture Requirements for Enhanced Billing Systems*, workshop presented at 2nd Annual Billing and Customer Service Operations in the Telecommunications Industry, Institute for International Research, New York, April 1996.

[2] Kenneth P. Birman and Thomas A. Joseph. *Exploiting virtual synchrony in distributed systems*. Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Austin, Texas, November 1987. pp. 123-138.

[3] A.V. Borshchev, Yu.G. Karpov, V.V. Roudakov. *COVERS - A Tool for the Design of Real-Time Concurrent Systems*. In: V. Malyshkin (ed.). *Parallel Computing Technologies. Proceedings of the 3rd International Conference PACT-95*, Lecture Notes in Computer Science No 964, Springer, 1995, pp. 219-233.

[4] A.V. Borshchev. *Modelling of Concurrent Real-Time Systems*. Ph.D. thesis, St.Petersburg State Technical University, April, 1995. *[in Russian]*.

[5] Jeremy J. Carroll and Andrei V Borshchev *Strongly Causal Time: Semantics, Implementation and Performance Modelling*, Hewlett-Packard Labs Technical Report HPL-96-XX, 1996 (HP labs TRs are available under http://www.hpl.hp.com/invented/techReports/)

[6] David R. Cheriton and Dale Skeen. *Understanding the Limitations of Causally and Totally Ordered Communication*. Proceedings of the ACM Symposium on Operating System Principles. 1993. pp. 44-57.

[7] Colin Fidge. Logical Time in Distributed Computing Systems. IEEE Computer 24(8), August 1991. pp. 28-33

[8] Keith Harrison, *A Novel Approach to Event Correlation*, Hewlett-Packard Laboratories Report, No. HPL-94-68, Bristol, U.K., 1994.

[9] David Hume. *An Enquiry Concerning Human Understanding*. Sect VII 'Of the idea of necessary connexion' part II. 1748.

[10]David R. Jefferson. *Virtual Time*. ACM Transactions on Programming Languages and Systems, 7(3), July 1985. pp. 404-425

[11]L. Lamport. *Time, Clocks and Ordering of Events in a Distributed System*. Communications of the ACM, Vol. 21,

No. 7, 1978, pp. 558-565.

[12]Friedmann Mattern. *Virtual Time and Global States of Distributed Systems*. In M. Cosnard and P. Quinton, (eds.), *Proceedings of International Workshop on Parallel and Distributed Algorithms (Château de Bonas, France, October 1988)*, Amsterdam, 1989. Elsevier Science Publishers B.V, pp. 215-226.

[13]P. Stephenson. *Fast causal multicast*. Ph.D. thesis, Cornell University, February, 1991