An application-level implementation of causal timestamps and causal ordering

Distrib. Syst. Engng 2 (1995) 74–86. Printed in the UK

An application-level implementation of causal timestamps and causal ordering

Andrew Berry[†]

Department of Computer Science, University of Queensland, St Lucia, Queensland 4072, Australia

Received 3 June 1994, in final form 8 May 1995

Abstract. Maintenance of causality information in distributed systems has previously been implemented in the communications infrastructure with the focus on providing reliability and availability for distributed services. While this approach has a number of advantages, moving causality information up into the view and control of the application programmer is useful, and in some cases, preferable. In an experiment at the University of Queensland, libraries to support application-level maintenance of causality information have been implemented. The libraries allow the collection and use of causality information under programmer control, supplying a basis for making causal dependency information available for application management and troubleshooting. The libraries are also unique in supporting existing distributed systems based on the remote procedure call paradigm. This paper describes the underlying theory of causality, and the design and implementation of the libraries. An event reporting service example is used to motivate the approach, and a number of previously unresolved practical problems are addressed in the design process.

1. Introduction

Lamport states [8]:

> In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation 'happened before' is therefore only a partial order of events in the system.

In contrast to this, events on a uniprocessor system (assuming no parallelism in the processor architecture) are totally ordered. This order can easily be determined, for example, by recording the processor time at each event. The 'happened before' relation (denoted '$\rightarrow$' in the remainder of the text) defined over a set of events in a distributed system determines the causal order of those events. Assuming a distributed system of several sequential processes, the $\rightarrow$ relationship is more formally defined as follows:

(i) If E_1 and E_2 are events in the same process and $Time(E_1) < Time(E_2)$, then $E_1 \rightarrow E_2$.
(ii) If E_1 is an asynchronous send event and E_2 is the corresponding receive event, then $E_1 \rightarrow E_2$.
(iii) If the pair (E_1, E_2) occurs in the transitive closure of 1 and 2, then $E_1 \rightarrow E_2$.

Any pair of events not related by the causal order are logically concurrent and cannot affect each other. A convenient visualization of this relationship is a space-time diagram, as depicted in figure 1.

Maintenance of causal ordering information has, in previous implementations (for example, Isis [1] and Psync [10]), been restricted to the communications infrastructure, with a focus on consistency and reliability. These systems ensure that messages between processes can only be delivered in a causal order, however, knowledge of causal dependencies is not passed on to the software developer. The use of multicast communication is also inherent in these protocols. While this approach provides transparency, correctness and efficiency for processes requiring causal ordering, knowledge and control of causality information by the application programmer (i.e. at the application programming level) can have distinct advantages.

In an experiment at the University of Queensland, prototype libraries to support the maintenance and use of causality information at the application level have been implemented. The libraries have been designed to support an event reporting service, thus the focus is shifted from providing reliability and availability, as is done by most existing implementations, to providing causal dependency information. The libraries are unique in supporting causality information for existing remote procedure call (RPC)-based distributed systems, using the ANSAware platform for the initial implementation. They are also not intrinsically tied to any communications protocol.

[†] e-mail: andybcs.uq.oz.au

This paper introduces the theory of causal ordering and timestamps, discusses the advantages of an application-level approach to maintenance of causal ordering information, and examines the design and implementation of libraries for maintaining this information.

2. Previous work

2.1. Causal timestamps

Fidge [4] has described a system of causal timestamps based on partially ordered logical clocks. These clocks can be used to determine the causal order of events in a distributed system. The method involves each process in the system keeping an array of logical clocks, one for each process in the system. The abstract data structure for storing this information is referred to through the remainder of this paper as a 'clock vector' [9].

Clock vectors, if recorded with events of interest, can be used to determine the $\rightarrow$ relationship between those events. The rules for maintenance and comparison of clock vectors are specified in [4], however, the following rules for maintenance of clock vectors are significant:

(i) Each process must increment its own logical clock in its clock vector when it performs an event.
(ii) An asynchronous message between processes must carry the current clock vector of the sending process. The receiving process(es) must merge the incoming clock vector with its own.
(iii) Synchronous communication between processes must result in the synchronization of their clock vectors (an example of a synchronous communication is an Ada rendezvous).
(iv) Dynamic creation of a process requires that a new logical clock be added to the clock vector. However, logical clocks cannot be removed from the clock vector—the termination of a process does not remove the need to store its logical clock.
(v) If a process is spawned by an existing process, then the child must inherit the clock vector of its parent to record the causal relationship between parent and child.
(vi) Logical clocks cannot be decremented.

The value of each logical clock in a clock vector represents the time at which the current process last had contact, either directly or transitively, with each other process. Figure 1 shows a space-time diagram for a distributed computation augmented with appropriate clock vectors. Note that arrows in the diagram indicate communication between processes and filled dots indicate events.

The following additional properties of clock vectors are useful in implementation and reasoning about the vectors:

(i) Zero-valued logical clocks in the clock vector (indicating no contact with the associated process) can be omitted. In effect, this means no logical clock is required for a process unless there is a causal relationship with that process.

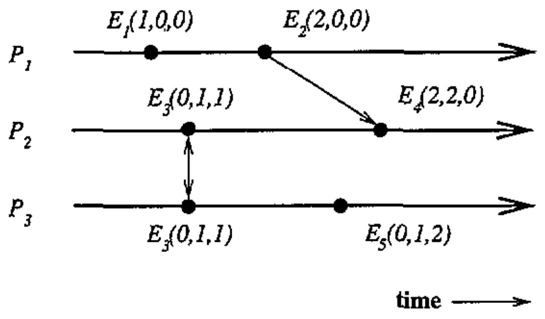

Figure 1. A space-time diagram augmented with clock vectors.

(ii) 'Events' can be defined to include only events that are of interest to the processes in the system, provided the rules for maintaining timestamps are not violated. For example, communication events need not necessarily lead to incrementing a logical clock.
(iii) It has been proven that a vector of length N is minimal to record the vector time for a static set of N processes. A number of optimizations of the method described by Fidge are possible, but these require additional storage and processing, and are not effective in all situations. A more detailed discussion of this issue can be found in [13].

The recording of causal timestamps is useful in many applications, particularly for debugging, event monitoring and detection of global states in distributed systems. For example, if a service in a distributed system fails, a record of events with their associated causal timestamps can be used to determine all events that could possibly have led to the failure. In particular, it allows us to immediately discard those events that do not causally precede the failure. Causal timestamps also provide the basis for determining a 'consistent cut' of a distributed computation. That is, the partitioning of the set of events into a 'past' and a 'future' set. Rules for determining a consistent cut are described in [4].

Causal timestamps suffer from a number of practical problems that limit their use in many systems. The two major problems are summarized below:

(i) In systems with many processes, the size of the vector can quickly become unworkable.
(ii) In theory, the size of individual clocks is unbounded for long-lived systems. A protocol for zeroing or re-use of clocks is necessary, for example, as described in [15]. However, such protocols generally lead to a loss of information about past events.

2.2. Causal message ordering

While causal timestamps allow us to determine the relative order of any pair of events, they cannot be used to determine if there are any intervening events. This means that messages can be delivered in an order that violates causality. Figure 2 shows such an ordering, where message M_1 is received after M_4, despite the fact that the send event E_1 which generated M_1 causally precedes E_4 which generated M_4.

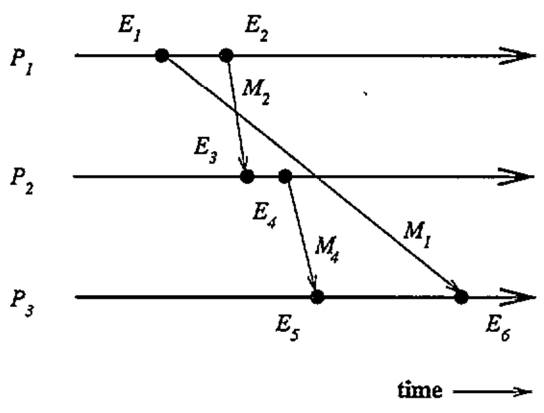

Figure 2. A message delivery ordering that violates causality

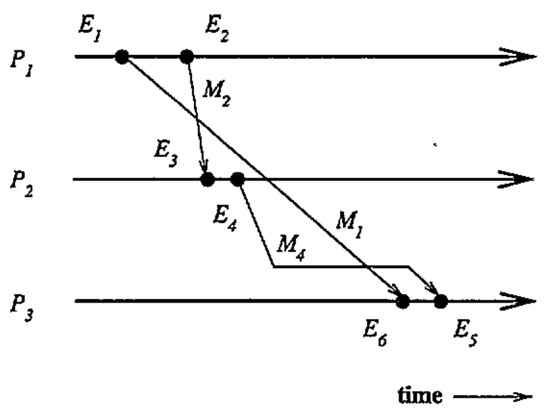

Figure 3. Message delivery with causal ordering applied

Causal message ordering guarantees that the order of delivery of messages does not violate causality in systems of communicating processes. Typically, messages are delayed at the receiver until all causally preceding messages are delivered. Figure 3 shows how figure 2 might look if causal message ordering was enforced.

A number of different mechanisms have been proposed and implemented to ensure causal message ordering:

(i) Schiper *et al* [11] propose an algorithm based on Fidge's clock vectors. Each process maintains a clock vector and a buffer containing the clock vectors associated with the set of undelivered messages, as seen by the current process. The current clock vector and the buffer must accompany each outgoing message as a timestamp. The destination process cannot accept a message until its own clock vector exceeds the clock vector of the most recent message addressed to that destination process recorded in the incoming buffer.
This algorithm increases the size of the timestamp from order N to order N^2 where N is the number of processes. However, optimizations detailed in the paper make the likely size of a timestamp somewhat smaller.

(ii) Raynal and Schiper [12] propose a modified implementation of the previous method that records only communication events. It has similar properties to the previous method, but is marginally simpler and more straightforward to implement. It loses generality though, since processes can no longer record the causality of events other than communication events.

(iii) Birman and Joseph [1] have taken a significantly different approach to implementing causal message ordering in the earlier version of the Isis toolkit. This algorithm requires that each communication carry a copy of all causally preceding messages that have not yet been acknowledged. If a process receives a message buffer containing several messages for the current destination, it simply processes them in the order that they appear in the buffer, which is guaranteed to be a causal order. Once again, this method does not allow the recording of events other than communication events.
The algorithm was implemented in the Isis toolkit as part of a reliable communications transport protocol based on multicast and process groups.

(iv) The current Isis algorithm [2] is based on the algorithm of Raynal and Schiper described in item 2 above, with a number of optimizations applied. The optimizations lead to a timestamp size of order N, and are made possible by the use of multicasting and process groups in Isis.
The major advantage of the current and previous Isis methods is that the handling of message and process failures are bundled with the ordering mechanism, providing a reliable transport protocol with well defined ordering semantics. The preceding approaches require modification or extension to cope with failures.

(v) Peterson *et al* [10] have devised and implemented the Psync communications protocol that supports causal ordering within the context of a 'conversation' (which is similar to the Isis process group). The protocol is efficient, but depends on all processes in the conversation receiving a copy of all messages, and does not preserve causality for communications in different conversations.

Causal message ordering is typically used to ensure the causal consistency of processes in distributed systems. This is particularly useful for multicast communication, and has been used in Isis as the basis for efficient replication algorithms. Causally ordered messages also provide a means to ensure the causal order of access to services. In everyday life, this is generally accepted as a 'fair' order. For example, if you told a friend you were on the way to the bakery to buy a cream bun, it would be rude for them to beat you there and take the last one. However, if you each decided independently to go to the bakery and buy a cream bun, there would be no conflict.

Despite its usefulness, there are still problems in the application of causal message ordering to distributed systems. Two of the major problems are:

(i) As with causal timestamps, none of the methods described is scalable to large systems. The size of the required timestamps is of order N (for multicast-based mechanisms) or N^2 for non-multicast mechanisms. In all cases, the methods become cumbersome as the number of participating processes grows, particularly when the communication patterns of individual processes involve contact with many other processes.

(ii) All methods require the participation of all processes in the system. There is no means of separating 'participating' and 'non-participating' processes, which can be useful for legacy systems or in Open Distributed Processing (ODP) systems, where communicating systems are often fully autonomous [6]. Isis and the Psync protocol achieve participation implicitly by building causal message ordering into network protocols.

3. High-level design issues

The primary design goal of the causality libraries is provide for the maintenance and use of causality information in a flexible and protocol independent manner within user applications. This section addresses the high-level design issues associated with these libraries. An event reporting service is introduced as an example to motivate the need for the libraries and to help justify a number of design decisions. In this section, the advantages and disadvantages of controlling causality information at the application level are discussed, and the practical problems of providing causality information for existing distributed systems are addressed.

3.1. An event reporting service example

Event reporting services have been provided or are being provided in a number of distributed programming environments, for example the OMG COSS event service [14]. These services collect event notifications from distributed applications and pass the notifications on to interested 'subscribers'. Typically, the events of interest are exceptional conditions, for example, failure, security violations or performance degradation.

One of the difficulties associated with collation and interpretation of events is determining dependencies between events. Event notifications are usually timestamped using the local clock of the originating process, but there are a number of problems with this approach:

- Real-time timestamps can only give a total ordering of events, rather than the partial order of those events.
- The inherent skew between distributed real-time clocks can lead to a false (i.e. non-causal) ordering.

The use of timestamps based on clock vectors can solve both of these problems, and would allow subscribers to an event service to ask, for example, 'when a failure of type X is reported, give me the last 25 events that causally precede the failure'. In the presence of large numbers of concurrent events, this capability would be extremely useful for management and troubleshooting purposes.

Causal timestamps, as described in section 2.1, can provide the capability discussed in the previous paragraph, with one restriction—it is not possible to determine if all causally preceding event reports have been delivered to the event service. For example, if an event report $E1$ is delayed, a causally succeeding failure $E2$ can be reported before $E1$, and subscribers interested in the events leading to $E2$ will not receive $E1$. This problem can be avoided by using causal message ordering, which will delay the delivery of $E2$ until $E1$ arrives. The choice between causal timestamps and causal message ordering is a cost/benefit trade-off, since causal message ordering involves significant additional overhead.

The need for causal dependency information is not always present, and some distributed applications will not require timestamps based on clock vectors. For this reason, it is useful allow applications to apply these mechanisms selectively.

3.2. Advantages of application-level support

As mentioned in section 1, there are some distinct advantages in maintaining causality information at the application level, rather than in the communications infrastructure:

(i) As can been seen in the event service example, it is sometimes necessary to make causality information accessible to the applications. Infrastructure implementations typically do not allow the application programmer to directly access causality information—they simply ensure that messages are delivered in a causal order. In this situation, it is not possible to determine the causal dependencies between messages within an application.

(ii) Applications using the infrastructure that do not require causal ordering carry additional, unnecessary overhead. An application-level implementation allows selective use of the mechanisms over a single infrastructure, although such use is subject to some restrictions to ensure consistency. These restrictions are discussed further in section 3.4.

(iii) Existing implementations provide causal ordering, where causal timestamps, which have lower overhead, are sufficient in some situations.

(iv) An application-level approach allows the programmer to decide which events are significant for causality, whereas infrastructure implementations can only record causality between communications events (i.e. message send and receive events). This means, for example, that a process can update its logical clock for a sequence of events, then report those events to an event service in a single message, rather than having distinct messages for each event.

(v) Implementation at the application level allows causal timestamps and ordering to be easily implemented over multi-protocol networks and existing distributed computing environments, for example OSF DCE or ANSAware. In other words, the implementation can be protocol independent.

The paper by Cheriton and Skeen [3] provides additional discussion of problems associated with implementing causal ordering in the communications infrastructure†. There are also some disadvantages of controlling such information at the application level:

† The Cheriton paper argues that implementations that support causality in the communications infrastructure are impractical. This author does not agree, seeing advantages in both approaches.

(i) If all processes require causal message ordering information, efficiency is improved by implementing the protocols in the communications infrastructure.

(ii) An infrastructure implementation is (mostly) transparent to the programmer and provides a programming model closer to that of traditional sequential programs, particularly in the Isis toolkit [2].

(iii) Infrastructure implementations are typically integrated with a communications protocol designed for reliability and availability in distributed applications. If these attributes are important for an application, an infrastructure implementation will be more effective and efficient.

In essence, the choice between application-level and infrastructure implementation is dependent on the requirements of applications. Where the focus is on providing flexible access to causality information, an application-level approach is superior. Where the focus is on providing reliability and availability in an efficient manner, an infrastructure implementation is more appropriate. However, an application-level implementation is essential to supply causality information for existing distributed systems that do not have protocol support. This author believes that this alone is sufficient argument for implementation of the libraries described in this paper.

3.3. Mapping messages to RPCs

All of the algorithms discussed in section 2 define their algorithms in terms of messages, and the algorithms are affected by the synchronous or asynchronous nature of communications. To apply the algorithms to RPCs, it is therefore necessary to map an RPC into a sequence of messages. There are two distinct possibilities:

(i) An RPC can be modelled as a single synchronous communication. In the pure and abstract sense, this is the most appropriate model for an RPC—an RPC is a single communication event. In practice, this is seldom the case, because processing an RPC takes a finite and sometimes significant amount of time, and because clients and servers are often multi-threaded. This means other activities can take place during the processing of a particular RPC, and there is considerable potential for loss of information regarding causality because of these factors. There is also the problem of nested RPCs—how do you model a 'nested' event?

(ii) The second, and more general mapping of RPCs is to map the request and response to distinct asynchronous messages. This has the following advantages:
   - The time taken to process the request is of no concern.
   - Non-determinism within client and server processes (due to threads) is easily handled. (See section 3.6 for a more complete discussion of this issue.)
   - Timestamps can be included simply as IN/OUT parameters to the RPC at the application programming level.
   - Non-participating processes can be accommodated with a weakening of the mapping (see section3.5).

   The main disadvantage is that this mapping does not capture the logical relationship between the request and response of an RPC. However, since the causal relationship between the communications is recorded, this is a minor criticism.

Based on the arguments above, our libraries are designed to support the asynchronous message mapping (although the synchronous mapping is not precluded). Given this choice of mapping for RPCs, the rules for maintaining causal timestamps and causal message ordering can be based directly on the algorithms discussed in section 2.

3.4. Interaction with non-participating processes

One of the difficulties associated with maintaining causality information is that communication between processes that occurs without using the infrastructure or libraries for maintaining causality can lead to unrecorded causal relationships. While this is undesirable, it occurs often in practice, since the size of timestamps required to record all causal relationships become impractical in large scale applications. In addition, it is sometimes necessary for processes that do not record causality, for example legacy applications, to interact with those that do.

There are two possible approaches to dealing with these problems—the application programmer can decide that such interaction will not adversely influence the consistency of causality information, or a restricted form of interaction that maintains causal consistency can be used. The theory for this restricted form of interaction has not, to the author's knowledge, been addressed in any existing literature, apart from Lamport's acknowledgment of the problem itself as a cause of 'anomalous behaviour' [8]. Rules for supporting such interactions are outlined in the following sub-sections.

Existing systems usually provide some means to restrict the scope of causality information being collected, leaving it to the application programmer to deal with the potential inconsistencies that might occur. The more recent version of the Isis toolkit [2] introduces *causality domains*, where communication across causality domains does not carry any causality information. Psync [10] restricts the transfer of causality information to a single *conversation*. While this allows the programmer to minimize timestamp size, there are no rules or support for ensuring the causal consistency of interaction with 'non-participating' processes.

One of the features of providing access to causality information at the application-level, is that the following rules for restricted interaction can be adhered to by application programmers to ensure causal consistency despite interaction with non-participating processes. The design of the causality libraries takes this into account, giving sufficient access to causality information to allow these rules to be implemented.

3.4.1. Causal timestamps For causal timestamps, the following rules can be used to ensure that the clock vectors maintained by participating processes are not invalidated by communication with non-participating processes:

(i) All processes for which causal timestamp information is maintained can receive asynchronous messages only from processes participating in the causal timestamping mechanism.
(ii) A process maintaining causal timestamp information can participate in synchronous communication only with other processes participating in the causal timestamping mechanism.
(iii) A process maintaining causal timestamp information cannot be spawned by a non-participating process.
(iv) A process maintaining causal timestamp information can send asynchronous messages to non-participating processes, and these messages need not (but are permitted to) carry timestamp information.

These rules are in addition to the rules for interaction with participating processes described by Fidge [4]. The rules imply that processes maintaining causal timestamp information can send messages to any other process, but cannot receive messages from processes that do not maintain causality information. For example, a process observing a group of processes that maintain causal timestamps can collect event notifications from the observed processes and determine their causal relationships without maintaining its own causal timestamp information. Note, however, that the causal relationship between events in participating processes and non-participating processes cannot be determined.

A further implication of these rules is that the logical clock associated with a non-participating process will always be zero in participating processes, and can therefore be omitted from timestamps. Informally, the rules are justified by asserting that a causal relationship between events $A \rightarrow B$, where A and B occur in distinct processes, can only exist if:

- A sends an asynchronous message to B
- **or,** A and B participate in the same synchronous communications event
- **or,** process A spawns process B
- **or,** there exists a sequence of any of the above actions that leads from process A to process B.

The only interactions that affect the causal timestamp information maintained by B are therefore:

- receiving a message
- participating in a synchronous communication
- being spawned.

By constraining these particular interactions so that they only involve participating processes, we ensure that the causal timestamp information remains consistent for the participating processes.

3.4.2. Causal message ordering

If a timestamp based mechanism is chosen for implementing causal message ordering, the rules for communication with non-participating processes described in section 3.4.1 can be applied to causal message ordering. In essence, processes maintaining causal message ordering information can ensure that the information remains consistent provided they do not accept messages from non-participating processes. The implications of this for non-participating processes are slightly different, however, and deserve further discussion.

A non-participating process cannot generally ensure that messages from participating processes are received in causal order. However, assuming the algorithm of Schiper *et al* [11], a process can guarantee the causal order of received messages through restricted participation in the ordering mechanism. In the following description, we will call a process implementing restricted participation an 'observer' process:

(i) The observer process must maintain a (single) clock vector reflecting the maximum of the clock vectors associated with all messages received from participating processes.
(ii) The fully-participating processes must record their interactions with the observer process (noting that in general, it is not necessary for participating processes to record their interactions with non-participating processes).
(iii) To ensure message ordering, the observer process must delay messages that carry a record of a message destined for the observer in their timestamp buffer if that timestamp is greater than the current vector clock.

3.5. Non-participating processes in RPC systems

The handling of non-participating processes is a difficulty for RPC systems, regardless of the mapping to messages chosen. This is because RPCs are inherently a two-way interaction, and the rules for communication with non-participating process described in section 3.4 only allow outgoing messages to non-participating processes. The problem can be solved by weakening the mapping of RPCs to messages as follows:

(i) RPC requests correspond to asynchronous messages.
(ii) RPC responses that have return values influenced by the semantics of the server correspond to asynchronous messages.
(iii) RPC responses that have no return values influenced by the semantics of the server can be ignored.

In essence, these rules state that provided the response to an RPC carries no information about the result of the request, it can be ignored when determining causal relationships. Note that this mapping could be further weakened in an application—if the response from an RPC is ignored by the application, then the causal relationship resulting from the response message can be ignored.

An implication of this rule is that communication failures can be signalled by a return value, whereas application failures or successes cannot. This leads to a causality relationship between the application and the infrastructure of the distributed system, an issue that is discussed in section 3.7.

3.6. Threads within processes

Most current distributed systems incorporate or support threads, which allow independent units of execution within a process. For example, in an RPC client process, a thread invoking an RPC is usually blocked while the RPC is being processed, but the remaining threads in the process can continue to execute†. In an RPC server process, multiple threads are used to provide concurrent processing of requests, provided appropriate resource locking is implemented.

The use of threads introduces concurrency to processes. In theory, it should follow that each thread maintains its own separate causality information—causality records all potential concurrency. The implementation of the libraries described in this paper does not preclude maintaining causality on a per-thread basis, but it is expected that users of the library will only maintain causality on a per-process basis. A per-process timestamp is assumed for the following reasons:

(i) Maintaining timestamps for each thread means that each thread must have a globally visible name, and must be addressable by that name. In most distributed systems, threads are anonymous, so this requirement cannot be satisfied.
(ii) Maintaining timestamp information for individual threads can significantly increase the number of logical clocks that must be recorded in the timestamp vector, hence increase the size of the vector clock.
(iii) Although there is concurrency between threads, there is a total ordering over the activities within the threads of a process (assuming a single-processor architecture). Provided updates to the timestamp of a threaded process are synchronized using a mutual exclusion lock, causality is not violated. Note that a similar argument has been used in [5] to justify maintaining only a single timestamp for each processor in a multi-processor system.

3.7. Infrastructure interactions

In most existing distributed systems, there are interactions with distributed infrastructure services inherent in the use of the system. For example, it is difficult or impossible to communicate with another process without first finding its address. In general, some form of name service is consulted to find the address.

These interactions result in a causal relationship between the infrastructure service and the process, and hence between processes sharing the service. Since it is generally difficult to add support for causality to the infrastructure of existing systems (access to source code is often not possible), these causal relationships cannot be recorded and a complete partial ordering of events in the distributed system cannot be determined.

However, from the perspective of an application programmer, the causal relationships resulting from interactions with the infrastructure do not generally affect the functional behaviour of the program. That is, interactions with the infrastructure are auxiliary to the purpose of an application—they simply support interactions with other application processes. For this reason, it is reasonable for applications to ignore such interactions when maintaining causality information, hence the libraries provide no direct support for recording interactions with the infrastructure.

† Note that blocking RPC systems without threads would have minimal need for causality information, since execution of RPCs is equivalent to execution of a local, sequential procedure call.

3.8. Failure handling for RPC systems

The handling of failure is particularly difficult for RPCs when causality information is being maintained. It depends particularly on the RPC semantics provided by the implementation. Unless there is some guarantee of execution (e.g. at most once semantics), it is difficult to characterize the causality relationship between the client and server when an RPC fails. At this stage, the issue of RPC failure has not been dealt with any further.

4. Low-level design and implementation

The causality libraries were implemented for the ANSAware distributed system which provides high-level RPC facilities in a Unix and TCP/IP environment. ANSAware was chosen because of its availability and its relatively clean and simple interface definition language (IDL) and distributed programming language (DPL). The ANSAware DPL is integrated with the C programming language, so C has been used in the implementation of the libraries. The libraries are largely independent of ANSAware, although there are some minor dependencies introduced to minimize the initial effort.

The current implementation is a 'proof-of-concept' version, intended to address the high-level design issues discussed in the section 3, and to allow us to gain experience in the use of causality information in distributed applications. Performance and implementation efficiency were not goals for this version.

The implementation is separated into two distinct libraries. A causal timestamps library provides a basic implementation of clock vectors based on the theory described in [4]. A causal ordering library provides and implementation of causal ordering based on the algorithm of Schiper et al [11]. Each library defines a data structure for storing the appropriate causality information, and a set of functions for creating, maintaining, transmitting and deleting this causality information. In effect, the two libraries each define an abstract data type.

Transmission of causality information is achieved by including a timestamp based on the abstract data type in the parameters of an RPC. Selective use of the mechanisms is achieved by either including or excluding these timestamps in the definition of interfaces supported by a given ANSAware program. A representation of each abstract data type is defined in ANSAware IDL so that it can be marshalled, transmitted and unmarshalled by automatically generated ANSAware code. Support for other RPC protocols can be achieved by modifying the libraries to generate appropriate representations of the abstract data types, or possibly by generating a platform independent representation that can be sent as binary data over any RPC protocol. The functions on each abstract data type are flexible enough to support the rules for interaction with non-participating processes.

4.1. Identifying processes

In order to correctly maintain causality information, all participating processes must be uniquely identified. To allow for some variation in the way processes are identified, the libraries simply use a variable length character string to store process identifiers. Some efficiency could be gained by using a less general structure, but it would place restrictions on the possible representations of process identifiers and reduce the protocol independence of the libraries. It is expected that system specific process identifiers would be used by applications, for example, DCE object identifiers.

4.2. Causal timestamp library

This library provides an abstract data type for the maintenance and use of clock vectors based on the theory described in [4].

4.2.1. Data structure In order to provide timestamps suitable for use with ANSAware RPC, a timestamp data structure was defined using the ANSAware Interface Definition Language (IDL). The structure is a variable length array of process clocks, with each clock containing a process identifier string and a positive integer clock value. The clock value is a 32-bit representation of an integer. The clock vector library uses this timestamp structure directly, adding only a mutual exclusion lock to complete the clock vector data structure. The mutual exclusion lock is necessary to deal with concurrency introduced by threads.

The clock value associated with the current process is always the first clock in the array. This ensures it can be quickly and easily accessed, and is trivial to implement.

4.2.2. Functions Six major functions were provided by the clock vector library, namely:

`vt_inc`: A function to increment the logical clock of the current process. This function increments the integer value associated with the current process in the clock vector.

`vt_merge`: A function to merge an incoming timestamp with the clock vector. This function applies the merge algorithm for vector clocks described in [4]. At present, finding matching entries in the two clock vectors is achieved using a simple linear search of the incoming timestamp for each entry in the local clock vector.

`vt_ts`: A function to create a timestamp suitable for use as an RPC parameter, copying the contents of the clock vector. It is necessary to copy the data structure to ensure the mutual exclusion between threads accessing the clock vector. Future versions might also modify the internal data structure to improve efficiency of merges, so this function is the place to deal with differences between internal representation and the RPC timestamp. If the library were to support multiple infrastructures (e.g. DCE and ANSAware) multiple versions of this function could be provided.

`vt_lock`: A function to lock the clock vector to ensure thread safety. All operations on the clock vector should be protected by a mutual exclusion lock.

`vt_unlock`: A function to release a lock on the clock vector.

`vt_before`: A function to determine if the $\rightarrow$ relation exists between two timestamps. This function does not operate directly on the clock vector, but two message timestamps. Typically, this function would be used by some event collection process to order events.

The functions to lock and unlock the clock vector are made available to the application programmer, rather than being implicit in individual functions. This is to allow related operations on the timestamp to be grouped without concern for interruptions by thread scheduling. For example, a thread might be interrupted between incrementing and copying the timestamp, potentially leading to an invalid timestamp.

A number of additional functions were implemented to assist in memory management of timestamps, and to create and delete the clock vector. These are not complex, so are not described in detail.

4.2.3. Using the library In order to correctly maintain information about causality between events, a set of processes must use the library in the following manner:

(i) Each process must have a unique process identifier, and this identifier must be supplied when creating the clock vector.

(ii) All RPC requests to processes maintaining a clock vector must carry a timestamp parameter.

(iii) A process maintaining a clock vector must not accept requests or return values from other processes unless a timestamp is provided as a parameter. Interactions with infrastructure functions, for example the ANSAware trader, are exempt from this rule.

(iv) When a timestamp is received by a process maintaining a clock vector, either as a parameter to a request or in the response to a request, the incoming timestamp must be merged with the clock vector of the process immediately.

(v) When a significant event occurs in the process, the vector clock must be incremented.

(vi) All operations on the clock vector must be protected by a lock.

(vii) All logically grouped operations on the clock vector must be protected by a single lock. For example, if the receipt of an RPC on an interface is a significant event, then the incoming timestamp must be merged and the clock vector incremented as a single logical step protected by a lock. If a copy of the current timestamp is required for reporting this event, then making the copy should also be part of the single logical step.

Figure 4 indicates how the library might be used, adding appropriate function calls to the space-time diagram of figure 1:

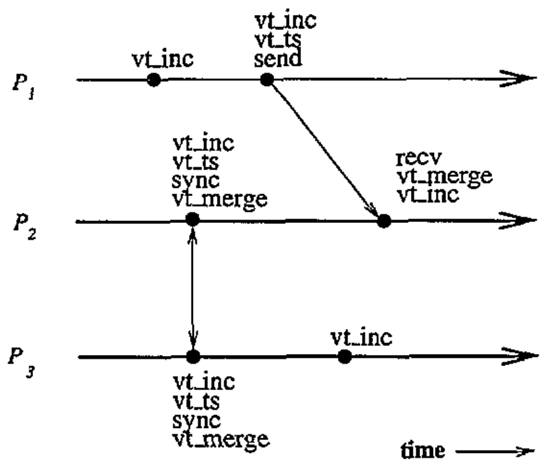

Figure 4. Using the causal timestamps library

Note that the calls to lock and unlock the vectors are omitted to keep the figure simple. To fully ensure thread safety, all of the function calls listed for each event should be protected by a (single) lock. However, in an RPC environment, it is difficult or impossible to separate the sending of the RPC from the response, and it is not acceptable to lock the clock vector for the duration of the RPC. In this case, the lock should be released immediately before sending the RPC. On the receiving end of an RPC, it is similarly difficult to lock the timestamp before receiving an RPC, so locking the clock vector immediately upon receiving the RPC is sufficient.

This weakening of locking semantics for RPCs means that the temporal order of communication events might not be reflected in the clock vector if two threads of a process are attempting to use the clock concurrently. However, the ordering implied by the clock vector will still be a correct causal order if no further processing within the current thread occurs between releasing or grabbing the lock and sending or receiving the RPC respectively.

The use of this library for an event reporting service is achieved by having the event service implement (at least) the restricted participation required of an 'observer process' described in section 3.4.1. The timestamps associated with reported events can then be used to create a partial order (an acyclic graph) of events. At the occurrence of an event that must be reported, an appropriate subgraph can be transmitted to the interested subscriber, noting that delayed event reports cannot be detected.

4.3. Causal message ordering library

The causal message ordering library was designed to provide the following functionality:

(i) Implement functions to manage a clock vector and timestamp buffer as described in [11]. The functions use the data structures supported by the causal timestamp library.
(ii) Provide a function to block the current thread until a message or RPC can be delivered without violating the causal order, based on the timestamp in the message or RPC.
(iii) Provide a mutual exclusion mechanism to ensure the thread safety of the clock vector and timestamp buffer.

Failure handling is not implemented in this version of the library. Future versions will most likely provide some means of recovering the causal ordering information when communication failures occur.

The algorithm of Schiper *et al* described in [11] has been chosen to implement causal message ordering because:

(i) The Isis algorithms and the Raynal and Schiper algorithm are unsatisfactory because they only support message ordering, and not the recording of causal relationships between arbitrary events.
(ii) The Psync algorithm was not chosen since it only allows the recording of causality in the context of a conversation.
(iii) The Schiper algorithm requires only a relatively simple extension to the basic causal timestamp mechanism.

4.3.1. Data structures

In order to provide message ordering timestamps suitable for use with ANSAware RPC, a timestamp data structure was defined using the ANSAware Interface Definition Language (IDL). This data structure is built on top of the timestamps defined for the causal ordering library, containing a clock vector for the current process and a set of timestamps representing the clock vectors of outstanding messages, as required by the algorithm of Schiper *et al* . The set of timestamps is represented as a variable length array of timestamp data structures from the causal ordering library.

The message ordering library uses a clock vector and a unordered, singly linked-list of timestamps as an internal representation. This internal representation is different from the timestamp to make the operations on the data structure more convenient—the linked list makes maintenance of the outstanding message list less troublesome than using the representation generated from the ANSAware IDL. The library also adds a mutual exclusion lock and a counting semaphore (event counter) allowing threads to block waiting for other messages to arrive. This data structure is referred to as the ordering clock in the remainder of the text.

4.3.2. Functions

The message ordering library implements only three major functions:

`ovt_send`: A function to generate a timestamp suitable for use as an RPC parameter based on the current ordering clock, and increment the ordering clock. Note that the increment occurs after the copy, as required by the message ordering algorithm. This function takes the internal representation and copies it into the external form for marshalling and transmission by ANSAware.

`ovt_deliver`: A function to block the current thread until a message with the supplied timestamp is deliverable, based on the rules of the message ordering algorithm. Blocking is achieved by waiting for the event counter of the ordering clock to be incremented. Once the message is deliverable, the incoming timestamp is merged with the current ordering clock and the event counter is incremented. The merging of clocks occurs as specified by the Schiper algorithm, with functions from the causal timestamp library being used to merge elements of the outstanding message list. Note that the matching of clock entries is once again implemented using a simple linear search on an unordered list.

`ovt_before`: A function to determine if the → relation exists between two ordering timestamps. This function does not operate directly on the ordering clock, but two message timestamps. Typically, this function would be used by some event collection process to order events.

Mutual exclusion is implicit in these functions, since the functions are logically complete, that is, users of these functions should not have any need to perform two or more operations that need to be grouped under a single lock.

4.3.3. Using the library In order to guarantee causal message ordering using the library, participating processes must abide by the following rules:

(i) Each process must have a unique process identifier, and this identifier must be supplied when creating the ordering clock. The current implementation simply takes a string parameter as the process identifier.
(ii) All RPC requests to processes maintaining an ordering clock must carry a timestamp parameter generated immediately prior to sending by the *ovt_send* function.
(iii) A process maintaining an ordering clock must not accept requests or return values from other processes unless a timestamp is provided as a parameter. Interactions with infrastructure functions, for example the ANSAware trader, are exempt from this rule.
(iv) When a timestamp is received, either as a parameter to a request or in the response to a request, the *ovt_deliver* function must be called before any processing of the request or response occurs. Due to the blocking nature of this function, processes implementing ordering clocks must be multi-threaded to avoid unbounded blocking. The number of threads available to a process must also be greater than the maximum expected number of messages awaiting delivery.

Figure 5 indicates how the library might be used, adding appropriate function calls to the space-time diagram of figure 3:

Notice the delay between the receipt and delivery of the message from P_2 to P_3. This delay is forced by the `ovt_deliver` function (the labelling on the diagram indicates function return rather than function call) to ensure causal ordering. Note also that the locking problems associated with RPC mechanisms outlined for causal timestamps in section 4.2.3 also apply to this library, although locking is not visible for this library.

In an event reporting service, this library could be used to delay incoming event reports until all causally preceding event reports have been delivered, ensuring the completeness of the event sub-graph that is sent to an interested process. It can be operated in the manner of an 'observer' as described in 3.4.2, using the ovt_deliver function to receive incoming RPCs, but without using the ovt_send function.

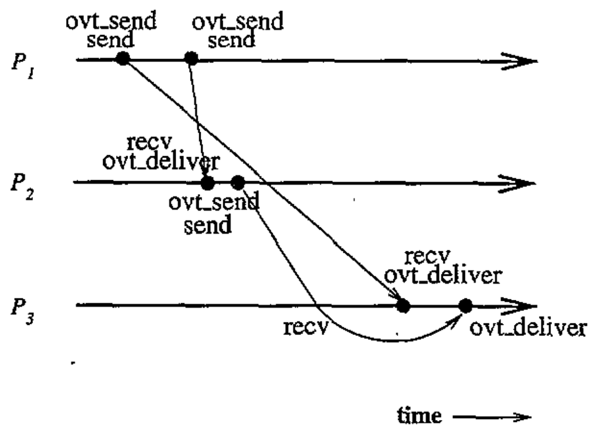

Figure 5. Using the causal ordering library

The implications of ignoring failure in this initial implementation are that failure can lead to unbounded blocking. Future versions of the library are likely to include some mechanism to prevent this problem.

4.4. Performance

Although the focus in the initial implementation is on functionality and flexibility, it is useful to provide performance figures to indicate the likely overhead associated with maintaining and transmitting causality information.

The test application was an Echo RPC, where a client sends an integer to a server and it is immediately echoed back to the client. Three different versions of the application were created:

(i) A simple version with no timestamps.
(ii) A version using causal timestamps.
(iii) A version using causal ordering.

All version accept a parameter indicating the number of RPCs to perform. The two versions maintaining timestamps also accept a parameter giving the size (number of process entries) in the timestamp, then generate arbitrary timestamps of that size before sending any RPCs. For the causal timestamps version, this resulted in timestamps of size $N * clocksize$, where N is the number of processes, and $clocksize$ is the size of the entry for each process. For the causal ordering version, this resulted in timestamps of size $2N * clocksize$. In theory, the size of a causal ordering timestamp can reach $(N * clocksize)^2$, however this size is unlikely to be reached in a typical application involving N processes. The likely size of timestamps depends on the communication patterns of the participating processes, so the size chosen for performance tests is arbitrary. Process identifiers in timestamps were represented in sixteen byte strings—the length of DCE object UUIDs.

There is no threading or concurrency involved in any version. This means, in particular, that there will be no delayed messages in the causal ordering version—the performance figures indicate the overhead associated with maintaining the information, rather than the likely delays associated with messages being delivered out of order.

Table 1.

	No timestamps	*Causal* timestamps	*Causal* ordering
Round Trip (ms)	3.8	4.9	5.4
CPU (ms)	2.3	3.1	3.2

The tests were carried out between a Sun Sparcstation 2 (client) and a Sun IPC (server), each running SunOS 4.1 and ANSAware 4.1 over a lightly loaded ethernet network. Two sets of performance figures are given:

(i) Round trip time (RTT) figures for an RPC, including all of the processing necessary to maintain and marshall the causality information.
(ii) CPU time per RPC at the client.

For the versions using timestamps, a range of sizes was tested, beginning from the minimum size (2). The baseline results (i.e. for the no timestamp version and the minimum timestamp size for the timestamp versions) are presented in table 1, showing the increase in round-trip and CPU time associated with simple use of the timestamp libraries.

The remaining figures are captured in the following graphs, which plot round-trip time and CPU time against timestamp size (number of process entries) for both causal timestamps and causal ordering.

The performance figures admit a number of interesting observations:

(i) Overhead associated with maintaining timestamps is significant, but not an order of magnitude larger, for small timestamps.
(ii) As timestamp size increases, both round-trip time and CPU usage grow in an approximately exponential manner, with an almost linear range for the region from approximately size 10 to size 100 for causal timestamps, and similarly for size 5 to size 50 for causal ordering.
(iii) Client CPU time is approximately half the round-trip time for larger sized timestamps. Assuming the server uses a similar amount of time to process the timestamps, this indicates that much of the round-trip time is spent dealing with timestamps and that efficiency improvements in the libraries are likely to be advantageous.

5. Discussion

5.1. Efficiency

Since implementation efficiency was not a major concern in the initial design and implementation of the libraries, there is scope for improvement in a number of areas. The performance figures indicate that attention to detail in the libraries could provide significant rewards. Some particular areas for investigation are:

(i) there are effectively three representations of a timestamp used in the system—an internal (library) representation, a C representation for marshalling, and the ANSAware wire representation. If the C representation generated by the libraries was protocol and platform independent, the timestamp could be transmitted in binary form thus removing the transformation to ANSAware wire format and resolving the problems associated with supporting multiple protocols.
(ii) the efficiency of the libraries could potentially be improved for large timestamps by using an indexed or sorted data structure to store timestamp vectors and outstanding messages sets. This would avoid the need for multiple, linear searches of data structures when merging timestamps.

5.2. Functionality

The functionality of the causal timestamp library is complete. It could perhaps be improved by removing the need for the programmer to explicitly lock the clock vector. This would involve providing a larger set of operations, including an *increment and merge* operation, and having each operation return the current value of the clock vector. The flexibility of the library would be marginally reduced by these changes.

A useful supporting library would be a set of functions for maintaining and traversing a directed acyclic graph that stores timestamps, with each edge representing a causal relationship between events. This would remove the need for an event reporting service to implement similar structures, and provide a basis for graphical output of event ordering information.

The functionality of the causal ordering library is minimal, although simple to work with. Failure handling is an obvious need, and some additional functions to access components of the structure might be useful for determining dependencies in an application. If desired, this implementation of causal message ordering could be incorporated into RPC stubs, which implement the RPC calling mechanism, thereby making causal message ordering transparent in an RPC environment. This application of the work could be implemented and tested with the ANSAware environment, since the source code is available.

5.3. Unresolved research issues

A number of areas of possible further research have been identified in this paper and during the implementation. The major issues are outlined below:

(i) The current implementations of the libraries are not scalable to larger systems or long-lived applications, since the timestamps can grow infinitely large, both in the number of process entries, and the size of the clock for each process entry. Some research into methods that reduce the size of timestamps is currently being carried out and will be reported in a subsequent publication.
(ii) Failure handling mechanisms for causal message ordering based on timestamps, particularly algorithms for rollback of updates to timestamp buffers, have not been described in detail or implemented. An application-level approach is required, but the failure

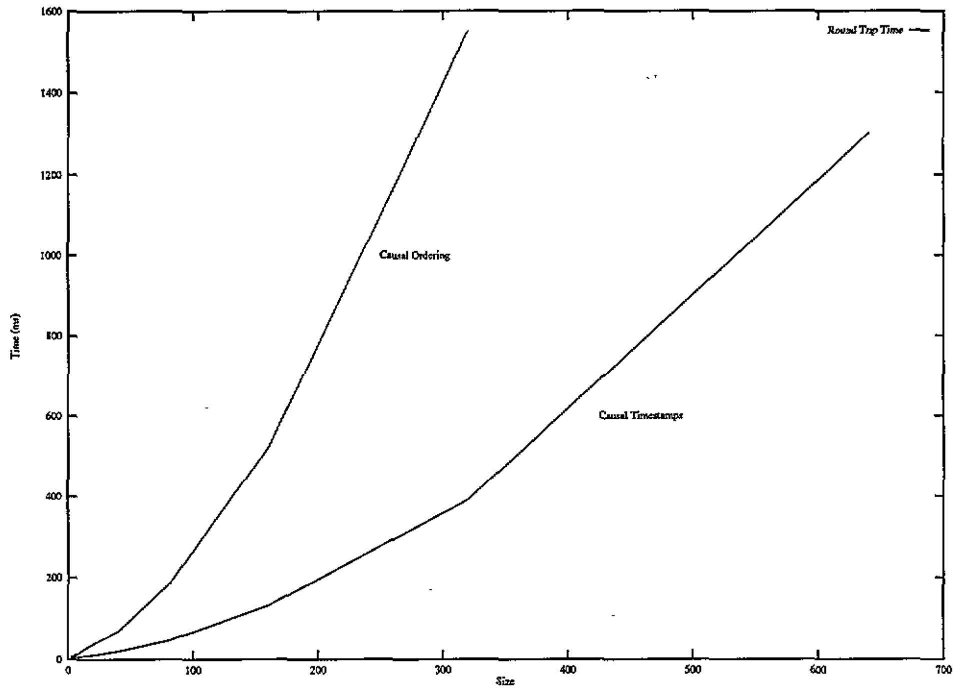

Figure 6. Round Trip Time for RPCs with Timestamps

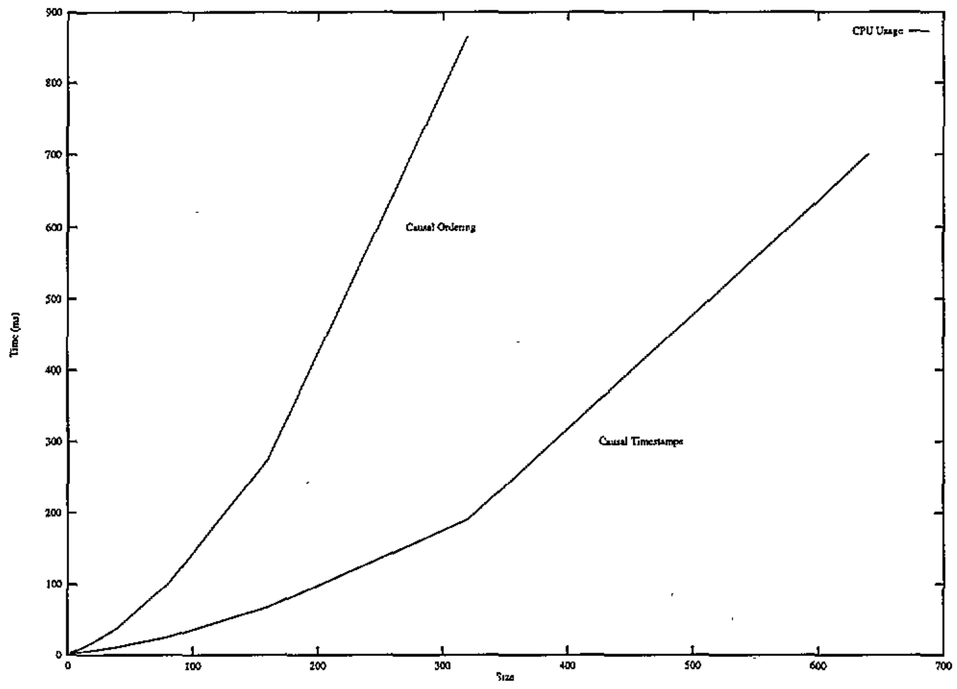

Figure 7. Client CPU Usage for RPCs with Timestamps

handling algorithms implemented for Isis [2] might provide some guidance.

(iii) The rules for interaction between participating and non-participating processes (with respect to causal timestamps or causal message ordering) should be formally proven and documented.

(iv) The causal relationships resulting from interaction with the infrastructure of a distributed system should be examined in the light of experience with application-level causal timestamps and causal message ordering.

5.4. Relationship to other work

This work is different from most existing implementation-oriented work involving causality information, in that it focuses on providing causal dependency information rather than providing reliability and availability. This is useful for application programmers, particularly in event monitoring applications. Comparison with work on causally ordered communication as in Isis [2, 1] and Psync [10] has already been addressed in section 3.2. In [7], an application-oriented use of causality is described, but it is also focused on the need for *reliability* and *availability*.

Much of the existing theory and algorithms associated with causality and causal ordering have been used in the design of the causality libraries. In particular, the work of Fidge [4] and Schiper *et al* [11] has provided the basis for the design and implementation.

6. Conclusions

This work has built upon the existing theory of causality and causal ordering to provide a general, protocol-independent implementation of causal timestamps and causal ordering that can support application programs in existing distributed programming environments. To the knowledge of the author, there is no other similar work.

In the design process, a number of previously unresolved practical problems have been addressed, particularly:

- the effect of threads within processes on the maintenance of causality information;
- rules for safe interaction with processes that do not provide causality information.

The resulting implementation is functional, providing sufficient functionality to support an event reporting service in the ANSAware distributed programming environment. The implementation is, however, relatively immature, with considerable potential for optimization, porting to other distributed programming environments, and further research.

Acknowledgments

I would like to thank Colin Fidge for his guidance during the project and for reviewing this paper. I would also like to thank Andrew Lister for his helpful comments. Thanks are also due to my employer, DSTC Pty Ltd, for use of a workstation and the ANSAware distributed programming environment.

References

[1] Birman K and Joseph T 1987 Reliable communications in the presence of failures *ACM Trans. Computer Syst.* 5(1) 47–76

[2] Birman K, Schiper A and Stephenson P 1991 Lightweight causal and atomic group multicast *ACM Trans. Computer Syst.* 9(3) 272–314

[3] Cheriton D R and Skeen D 1993 Understanding the limitations of causally and totally ordered communications *Operating Syst. Rev.* 27(5) 44–57

[4] Fidge C 1991 Logical time in distributed computing systems *IEEE Computer* August 28–33

[5] Haban D and Weigel W 1988 Global events and global breakpoints in distributed systems *Hawaii International Conference on Systems Sciences* (Los Alamitos, CA: IEEE) pp 166–182

[6] ISO/IEC JTC1/SC21/WG7 1993 Draft recommendation x.901: basic reference model of open distributed processing—part 1: overview and guide to use *Working Document ISO/IEC JTC1/SC21/WG7 755* (International Standards Organisation)

[7] Ladin R, Liskov B, Shrira L and Ghemawhat S 1992 Providing high availability using lazy replication *ACM Trans. Computer Syst.* 10(4) 360–391

[8] Lamport L 1978 Ttime, clocks, and the ordering of events in a distributed system *Commun. ACM* 21(7) 558–565

[9] Mattern F 1989 Virtual time and global states of distributed systems *Parallel and Distributed Algorithms* ed M Cosnard and P Quinton (Amsterdam: North-Holland) pp 215–226

[10] Peterson L, Buchholz N C and Schlichting R D 1989 Preserving and using context information in interprocess communication *ACM Trans. Computer Syst.* 7(3) 217–246

[11] Schiper A, Eggli J and Sandoz A 1989 A new algorithm to implement causal ordering *Distributed Algorithms (Lecture Notes in Computer Science)* 392 ed J C Bermond and M Raynal (Berlin: Springer) pp 219–232

[12] Schiper A and Raynal M The causal ordering abstraction and a simple way to implement it *Technical report 501* Institut de Recherche en Informatique et Systemes Aleatoires, Campus Universitaire de Beaulieu, 35042 - Rennes Cedex, France

[13] Schwarz R and Mattern F 1992 Detecting causal relationships in distributed computations: in search of the Holy Grail *SFB 124-15/92* (University of Kaiserslautern)

[14] Siegel J (ed) 1994 *Common Object Services Specification* vol 1 (Object Management Group) ch 4 pp 31–64

[15] Singh A K 1992 Bounded timestamps in process networks *Preprint* (Department of Computer Science, University of California at Santa Barbara)