CAUSALLY-STRUCTURED OUTPUT STREAMS FOR DISTRIBUTED COMPUTATIONS

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science with Honors in Computer Science from the College of William & Mary in Virginia,

by

Neal Douglas Cardwell

Accepted for ____________________

____

____

____

Williamsburg, Virginia
May 1996

Abstract

Output streams are an essential tool of sequential programming. Unfortunately the output stream abstraction cannot be easily extended to distributed systems because output streams require an ordering scheme, and there exist many competing ordering schemes for distributed systems. This thesis presents the alternative ordering schemes for distributed systems and proposes a simple definition of distributed output streams structured by causality. Also presented are extensions to the PVM distributed programming environment that provide an implementation of causally-structured output streams. A tool is described that provides a graphical view of output streams generated in a PVM computation. Finally, an example shows how causally-structured output streams and the stream viewing application can be used to debug a distributed application with a synchronization error.

Contents

List of Figures

Chapter 1

Introduction

1.1 Output Streams in Sequential Programs

Output streams are ubiquitous in computing. From the user's perspective a typical output stream is a list of the files in a directory, or a display of the text in a file. From a programmer's perspective, typical output streams include the `stdout` and `stderr` streams from the C standard library and the `cout` and `cerr` streams from the C++ standard library. From a system's perspective, a typical output stream runs from one UNIX tool through a pipe to another tool and then through another stream to a terminal.

But the concept of output stream is much more general. A *sequential output stream* is any sequence of *entries* written in the same sequential order. Entries near the beginning of the sequence were written earlier in time; entries later in the sequence were necessarily written later. To formalize this intuitive point, one can define a total order $\Rightarrow$ over all entries in the stream such that two stream entries e_1 and e_2 are related as $e_1 \Rightarrow e_2$ if and only if e_1 was written to the stream before e_2. The output stream itself is then the set of entries together with the $\Rightarrow$ relation.

Typically, the entries in an output stream are bytes representing ASCII-encoded human-readable text, but this is not necessary. An entry might be a 4-byte binary integer, a byte of compressed text, or a single byte of 8-bit audio data.

An output stream, and the bytes in it, have a source and a destination. The source is nearly always a process and the destination is typically a terminal, a virtual terminal window, a printer, a sound port, a file on disk, a serial port, or another process.

Sequential output streams have many nice properties:

- They are simple, standard, and widely available. Nearly every language and run-time environment provide a simple stream abstraction that can output text to a terminal, file, printer, or destinations.

- The concept of a stream is very general, since it is appropriate for output to many kinds of devices. More complicated structures – records, lists, or the structures used by database applications – are difficult to use in as many contexts.

- They provide a simple and consistent method for generating a program's output. Most UNIX and DOS tools, for example, accept input from text streams and write their output to text streams.

- During debugging, diagnostic stream output can provide a quick summary of the state of the program, including arbitrarily complicated data structures, without the need to stop the program in a debugger.

- Streams can provide a temporally-ordered log of events over the course of the entire computation.

- Testing tools can check a program's output stream against a file containing the expected output.

The last three properties are particularly important when debugging. Though considered very "low-tech," debugging programs by printing information to output streams is very popular and quite effective. Because the current study of output streams was motivated by a desire to use output streams for debugging distributed systems, this bias toward debugging examples will persist.

1.2 Output Streams in Asynchronous Distributed Computations

Distributed computations involve processes dispersed across a network communicating solely via message-passing. The machines on which the processes are executing share neither memory nor a clock. Consequently the activities of a process in a distributed computation can be broadly categorized as message sends, message receives, and internal computations.

Ideally, output streams could be employed easily and effectively in distributed systems exactly as in sequential systems. Unfortunately there are several obstacles to overcome before output streams are useful, or even well-defined, in distributed systems as they are in sequential systems.

First, the output streams produced by the dispersed processes must somehow be combined. The current standard operating procedure is for the programmer to manually combine the output of several processes by inspecting several trace files or the output displayed on several XTERM, TELNET, or debugger windows. This is a cumbersome and error-prone process, impractical for systems with many processes or large volumes of output.

Second, even after the output is combined, the user will often want to know which process produced which entries in the stream. For example, if process P_1 writes the string "do" to a stream and subsequently process P_2 writes the string "nut" to the same stream, the user will likely want to know that the resulting string "donut" was the product of the catenation of two separate strings – "do" and "nut."

Third, when entries from several processes are combined, the programmer would ideally like to see the entries ordered chronologically by the time at which they were produced. That is to say, just as in streams on conventional uniprocessor or shared memory systems, one would like the entries to be ordered strictly chronologically by the $\Rightarrow$ relation.

Unfortunately it is impossible to produce such an order in an asynchronous distributed system, since the clocks in such a system cannot be synchronized with absolute precision. It turns out that there are several alternative ordering schemes for distributed systems. The challenge is to find the order that most usefully characterizes the course of the distributed computation generating the stream, and then to find a clear and effective means of creating and displaying the stream subject to this order.

1.3 Related Research

The function of distributed output streams is to collect and organize the output of a distributed computation so that the relationships between the output data are clear and reflective of the course of the computation. By contrast, most previous research in the area of input and output for parallel and distributed systems has focused on high-performance, scalable, and fault-tolerant file systems. More pertinent to this effort is the large body of research in the area of debugging, monitoring, and visualizing distributed systems.

Debugging, monitoring, and visualizing schemes for distributed systems can be broadly categorized as "live" or "post-mortem," depending on whether the monitoring takes place as the computation unfolds or after the computation is finished. If a system involves live interaction with the computation then there is usually a central monitor to which processes report their status. If the interaction with the computation is post-mortem then often each process will write information about events and state to a per-process trace file, which the debugging or visualization tool will then read.

Many of these tools for distributed computations provide a graphical representation of the messages sent and received in the system. Topol, Stasko, et. al [13] describe a visualization tool that uses an extension to PVM, the popular Parallel Virtual Machine message-passing library, to automatically produce data to drive a visualization. Their system uses Lamport logical clocks for causal ordering and local real-time clocks for performance analysis. Edwards and Kearns [2] describe a distributed trace visualization system that uses per-process trace files with vector time-stamped information to provide post-mortem space-time diagrams as well as variable views and predicate evaluation. Joyce et. al [6] describe a tool that provides live textual and graphical display of events in a distributed computation.

Other systems a post-mortem replay of the execution of the distributed system. R. LeBlanc et. al[8] present a system featuring graphical animation of events such as process creation and termination and message sends and receives.

Some tools provide complete interactive debuggers that allow programmers to stop processes, investigate the state of the computation, step processes, set breakpoints, and restart the processes. Some of these systems are essentially off-the-shelf debuggers attached to each process in the computation; others are systems custom-tailored to debug distributed systems. Some even [12] perform run-time assertion checking.

None of this extensive work on parallel and distributed I/O, monitoring, debugging, and visualizing mentions extending output streams to the context of a distributed system.

1.4 Using Causally-Structured Output Streams as a Debugging Tool

To solve the problems in using stream output in distributed programs we extend the traditional idea of sequential streams by introducing the notion of causally-structured output streams for distributed systems. In order to prove the feasibility of this concept and explore unforeseen problems in implementing such a construct, causally-structured distributed streams were implemented in the context of the PVM distributed programming environment.

There are three facets of this effort to provide causally-structured distributed streams:

1. Extensions to PVM to provide a logical clock for each PVM task in the computation.

2. Extensions to PVM to provide a facility for writing output to causally-structured distributed streams.

3. A tool to provide the programmer with a detailed, flexible, useful view of output streams in a PVM computation.

In contrast to existing distributed system monitors and visualization tools, this system focuses on organizing the *output* of the computation, not displaying or debugging the communications or internal computation. A programmer can, if he or she wishes, write to a causally-structured output stream extensive information about the state of the program or a detailed narrative of the communications or computations in which a process is involved. But the programmer has complete control over *exactly* what is displayed; there is no *automatic* logging of every single message or process creation.

In addition, this system allows programmers to separate the output of a distributed computation into different output streams. Just as sequential programs commonly separate their output into two streams according to whether the output is normal or due to errors, distributed computations are able to separate their output into any number of causally-structured output streams based on the source or the semantic content of the information.

Allowing the programmer to specify what information ends up in a stream, and to which stream information is written, will help eliminate the problem that visualizations of distributed and parallel computations are often so busy and cluttered that they resemble an "angry fruit salad on drugs" [10] and are thus very difficult to use to use to learn anything useful about the system.

1.5 Organization

Chapter 2 offers a brief introduction to PVM. Chapter 3 describes the causal relation used to order events and explains why this ordering scheme is the best alternative. Chapter 4 outlines the design of the facilities added to PVM to keep track of the causal relationships in the system. Chapter 5 describes the causally-structured distributed stream abstraction and its implementation, and chapter 6 describes the use and implementation of the stream-viewing tool. Chapter 7 provides a case study of the use of dynamic vector time, output streams, and the stream viewer tool to debug a program. Chapter 8 wraps up with a summary of the results of this research.

Chapter 2

PVM

2.1 Description

PVM, Parallel Virtual Machine, is a software library that allows a heterogeneous collection of networked computers to be used as a single computing resource – a parallel virtual machine [5]. PVM is a standard, full-featured, well-documented library well-accepted in the research community for use in scientific computation and other computationally-intensive applications. PVM programs can be written in C, C++, or FORTRAN and may be easily re-compiled for a number of platforms, including most UNIX-like workstation operating systems and many shared-memory and message-passing supercomputers.

PVM provides facilities for message-passing, task creation, adding and removing hosts from the virtual machine, group communication and synchronization, and the use of load-balancers and debuggers. A bit of a warning: PVM refers to processes in the PVM virtual machine as "tasks," so the terms are used interchangeably here.

2.2 Alternatives to PVM

One alternative to using a message-passing library such as PVM would be to program directly to inter-process communication facilities provided by the UNIX kernel. A message-passing library was chosen instead because this enables changes to be made without recompiling the kernel and rebooting the machine. This also allows the use of a standard user-level debugger during development.

PVM was chosen instead of the Message-Passing Interface (MPI), another popular message-passing environment, [14] because it lacks PVM's facilities for dynamically adding tasks and hosts to a computation.

2.3 Implementation

Every host in the PVM virtual machine has a PVMD daemon executing on it. This daemon handles the configuration of the machine and forwards the messages from PVM tasks to the PVMD's on other machines. There is a single "master PVMD" in a PVM virtual machine that handles the addition and deletion of hosts from the virtual machine and other global configuration operations. Task-to-task, task-daemon, and daemon-daemon communications take place over TCP (and UNIX domain) sockets.

Though PVM is a multi-lingual and multi-platform library, this research was carried out on SunOS and Linux single-processor machines and in C and C++.

Chapter 3

Logical Time in Dynamic Distributed Systems

3.1 Alternative Ordering Mechanisms

As mentioned in chapter 1, sequential output streams are ordered by a simple total order $\Rightarrow$ that orders entries chronologically. The challenge in defining output streams for distributed systems is choosing an order for the entries in the stream. Unfortunately, for streams generated in a distributed computation there are a variety of choices for the ordering scheme, each of which has different characteristics. Fidge [4] provides a comprehensive taxonomy of these alternative ordering mechanisms, with a characterization of the faults of each.

Most ordering mechanisms for distributed systems are defined in terms of events. An *event* is a discrete action in some process in a distributed system. At a minimum this includes the creation and termination of pro-

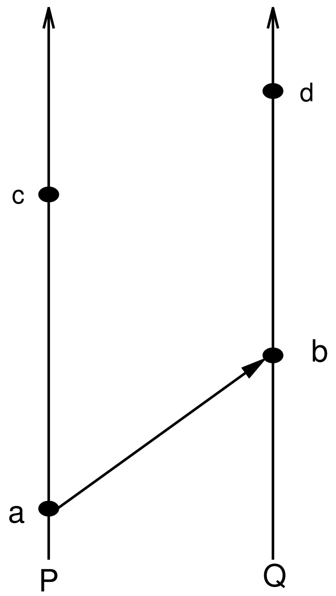

Figure 3.1: A Space-Time Diagram

cesses and the sending and receiving of messages. But any discrete action the programmer finds significant – such as a function call or an assignment to a variable – may be considered an event.

One standard way to graphically depict the events in a distributed system is a space-time diagram [7]. Figure 3.1 presents an example. In a space-time diagram events are placed according to the process in which they occur and the absolute real time at which they occur. Processes are depicted as vertical lines and events are dots on these lines. Time increases along the vertical axis. Messages are denoted by arrows from the source to the destination.

There are two main approaches to ordering events in a distributed system. The first is to use real clocks, such as the clock commonly included in a personal computer or workstation's motherboard. The second is to use a scheme that orders events only relative to each other. Such a time scheme is often called a *logical time* scheme, to distinguish it from a scheme dependent on real clocks.

3.2 Ordering via a Central Monitor

One of the easiest methods of ordering events in a distributed system is to send a notification of every event to a central logging or monitoring process. Events are then ordered by the order in which their notifications reach the observing process. This approach suffers from several problems, as noted by Fidge [4].

First, the ordering of events may be incorrect. Consider the situation in figure 3.2. In this example the central monitoring process, R, perceives event b to precede event a, even though these events occur in the opposite order.

Another problem this scheme suffers from is that the same computation, when executed several times, will likely have a different observed event ordering due to variability due to the effect of system load and network traffic on the sending of the notification messages. For example, the same computation as in figure 3.2 could result in the observed ordering in figure 3.3, where the central monitoring process this time perceives a to precede b.

And not only the notification messages are prone to different orders; events in the computation itself may occur in different orders. Consider figure 3.4, which depicts another execution of the same computation as figure 3.1. In figure 3.4 d precedes c due to a heavier load on the processor on

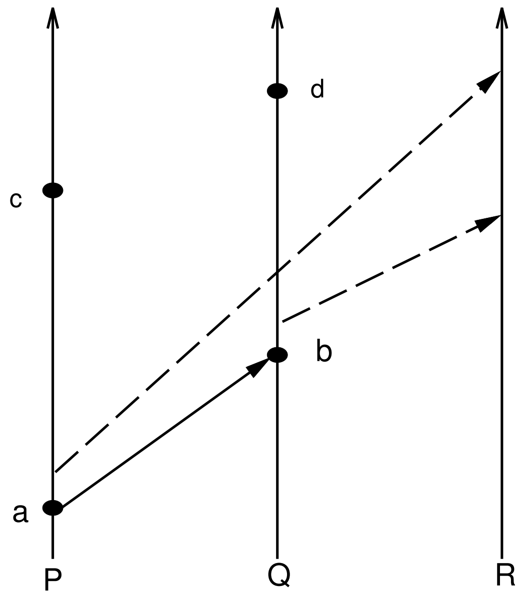

Figure 3.2: Incorrect Ordering with a Central Monitor

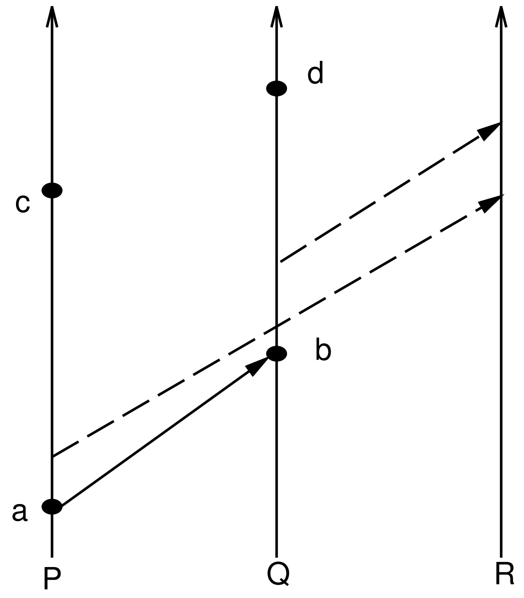

Figure 3.3: Variations in Perceived Ordering with a Central Monitor

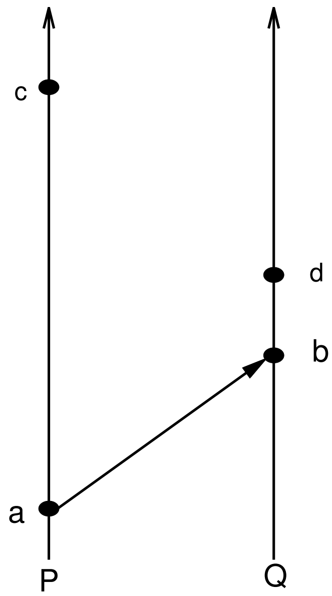

Figure 3.4: Variations In Event Order

which P is executing. But this monitoring approach is ill-equipped to deal with this possibility, for it will only report the (arbitrary) ordering as it occurred in a particular execution. It can tell an observer nothing about other possible interleaving of events. The only way to explore the different possible interleavings is to re-execute the program repeatedly. This is a serious liability when trying to reproduce a concurrency bug in a distributed system.

This scheme also suffers from the *probe effect*. A probe effect occurs when a monitoring scheme alters the course of the computation it is monitoring. This is a significant factor with a central monitoring process because of the extensive message passing involved in notifying the central monitor of every event. This message passing will slow down the computation, altering both the absolute and relative ordering of events in the system. This is particularly troublesome if synchronous message passing is used to communicate with the central monitor process. Using synchronous message passing will cause the monitored processes to block while they wait for an acknowledgment from the monitor process.

3.3 Approximately Synchronized Clocks

Of the ordering schemes that use real clocks, the most reasonable is the use of approximately synchronized clocks. The idea here is for each process in a distributed computation to have a real clock that is occasionally synchronized to within a tolerance τ of the clocks of other processes in the system. This requires a significant amount of message-passing but allows for an intuitive, straightforward notion of ordering.

The biggest problem with this scheme is that given any realistic value of τ, there will be the likelihood of two events in the system occurring "too close together." This occurs when two events occur, one at time t_1 and the other at time t_2, such that $|t_1 - t_2| < \tau$. When events occur this close together (such as events c and d figure 3.5) there is no way to determine which event actually occurred first. The imprecision of the clocks hides the exact order. This may seem like an unlikely occurrence, but with current processor technology events can happen within microseconds of each other and with current network technology clocks are likely to be synchronized, at best, to within milliseconds.

This approach also suffers from the same problem as the previous ap-

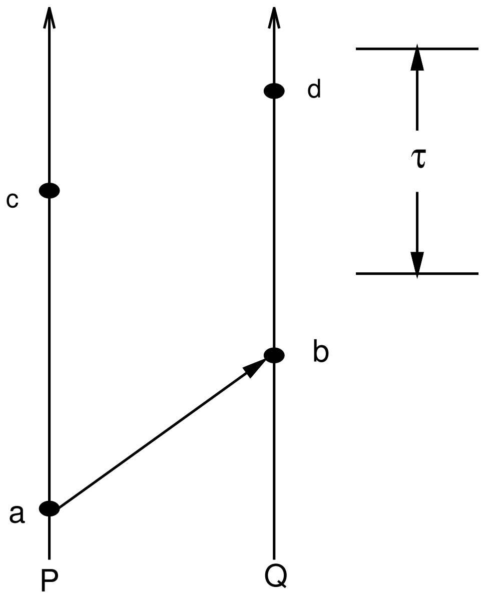

Figure 3.5: Clock Precision and Event Ordering

proach with respect to the adoption of the arbitrary event ordering of a computation. Real time stamps on events offer little insight into the possible variations in order that can occur between several repetitions of a computation.

3.4 The Happened-Before Relation

The most promising method yet discovered to order events in a distributed system is to order them according to their causal relationship. The causal structure of a distributed system is most appropriately described by the "happened-before" relation, commonly denoted by $\rightarrow$, defined by Lamport [7]:

Definition 3.1. *The relation* $\rightarrow \subseteq \mathcal{E} \times \mathcal{E}$ *on the set* $\mathcal{E}$ *of all events in a distributed system is the smallest relation satisfying the following conditions:*

1. *If* e *and* e' *are events in the same process, and* e *occurs before* e'*, then* $e \rightarrow e'$*.*

2. *If* e *is the sending of a message by one process and* e' *is the receipt of the same message by another process then* $e \rightarrow e'$*.*

3. *If* $e \rightarrow e'$ *and* $e' \rightarrow e''$ *then* $e \rightarrow e''$*.*

So defined, the $\rightarrow$ relation is an irreflexive partial order on the events in the system. Intuitively one may understand the statement $e \rightarrow e'$ to mean that e *could possibly* have a causal effect on e'. If $e \not\rightarrow e'$ and $e' \not\rightarrow e$ then e and e' are *causally independent* or *concurrent.* Two concurrent events cannot possibly affect each other, and so could happen in either order. If e and e' are concurrent we denote this by $e \parallel e'$.

Figure 3.6 provides an example of the $\rightarrow$ relation. In this diagram, for example, $a \rightarrow b$, $b \rightarrow g$, and $f \rightarrow h$ because these events are in the same process and the first occurs before the second (rule 1). Three pairs of events are related because one is a message send and one a message receive (rule 2): $a \rightarrow b$, $f \rightarrow e$, and $g \rightarrow h$. Finally, many pairs of events are causally related because of transitivity (rule 3), including $a \rightarrow e$, $a \rightarrow g$, and $c \rightarrow g$.

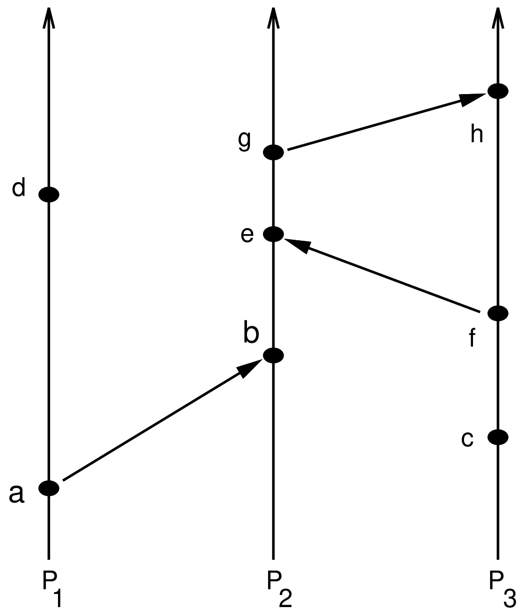

Figure 3.6: Causality

3.5 Mattern's Logical Time

Mattern [9] suggested an ordering scheme that relies on a vector-valued logical clock at each process to order events according to their causal relations. The relation between the vector-valued time stamps is isomorphic to the happened-before relation.

3.5.1 The Structure and Implementation of Vector Time

In Mattern's vector-based logical time model the current global time for a system of n processes – available, of course, only to an imaginary external observer – is a vector V of n non-negative integers, where $V[i]$ is the number

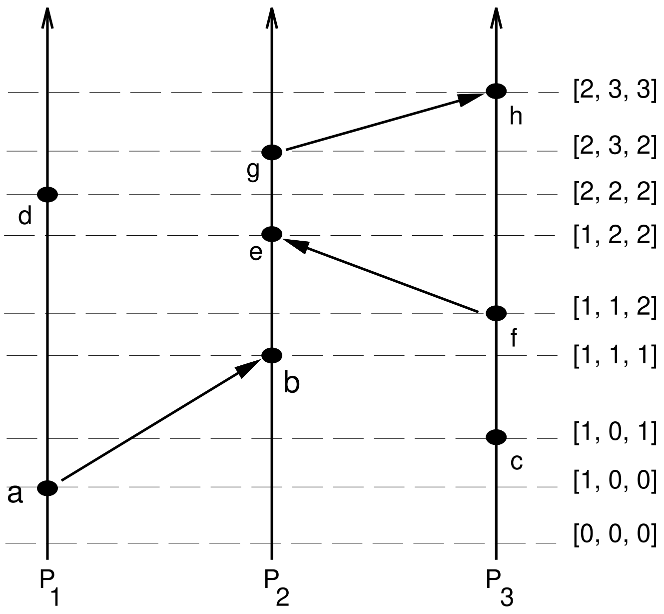

Figure 3.7: Global Vector Time

of events that have occurred in process P_i. [1] Figure 3.7 illustrates how this global time vector evolves as the computation unfolds.

Each process P_i maintains a logical clock, C_i, that has the best possible approximation to this global logical time, given only the information P_i can obtain from the messages it receives in the normal course of the computation. Like the global time it approximates, C_i is a vector of length n. The elements of C_i represent P_i's knowledge about the occurrence of events in other processes. The larger the value of $C_i[j]$, the more of P_j's events P_i is aware of, and the closer C_i is to the actual global logical time.

The rules for implementing vector time are simple:

[1]Strictly speaking the elements of the vector can be integers, naturals, rationals, or reals, and the value of each element need not increase by one.

- Process P_i's logical clock is initialized as follows:

$$\begin{aligned} C_i &\leftarrow\ < 0, 0, \ldots, 0 >; \\ C_i[i] &\leftarrow\ 1 \end{aligned}$$

- Before each event in P_i the local clock, C_i, "ticks" by incrementing the local component:

$$C_i[i] \leftarrow C_i[i] + 1$$

- P_i piggybacks on every outgoing message the current value of C_i.

- When P_i receives a message with a time-stamp V it updates its approximation of the progress of processes in the system:

$$C_i \leftarrow MAX(C_i, V)$$

  where MAX is the component-wise maximum of its two vector operands.

Because process P_i is the only process that can update the i-th element of a time vector, P_i always has the most recent knowledge of its own time. Process P_i accumulates knowledge about the occurrence of events in another process, P_j, either directly, from messages from P_j, or transitively, from messages from other processes that have gained direct or indirect knowledge about P_j.

Figure 3.8 shows a computation with the value of each process's local clock at each event, as well as the time stamp values for each message.

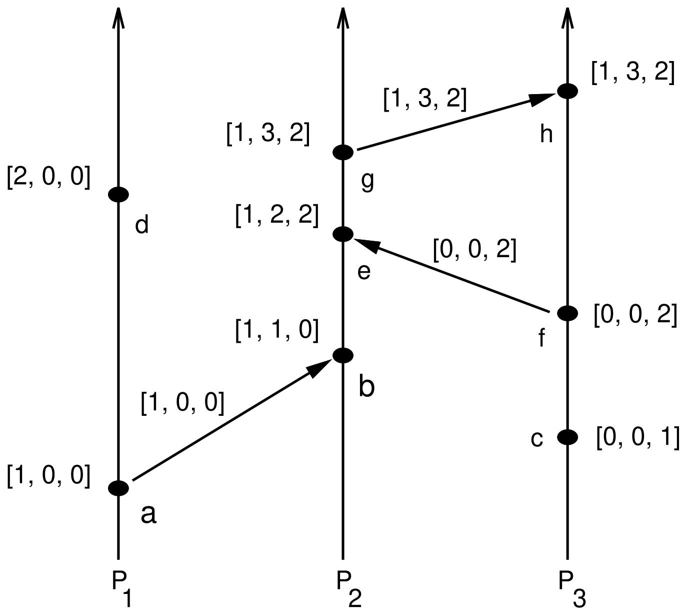

Figure 3.8: Vector Time

3.5.2 Causality and Vector Time

Mattern defines three relations on time vectors:

Definition 3.2. *For two logical time vectors U, V*

- $U \leq V \iff \forall i : \; U[i] \leq V[i]$
- $U < V \iff (U \leq V) \wedge (U \neq V)$
- $U \mid V \iff \neg(U < V) \wedge \neg(V < U)$

Vector time has the very useful property that these relations are isomorphic to the causal relationships between events:

Theorem 3.3. $\forall\, e, e' \in \mathcal{E} :\ e \rightarrow e' \iff C(e) < C(e')$

Theorem 3.4. $\forall\, e, e' \in \mathcal{E} :\ e \parallel e' \iff C(e) \mid C(e')$

Given these two theorems, all we have to do to determine the causal relationship between two events e and e' is to check their time stamps to see whether $C(e) < C(e')$ or $C(e') < C(e)$. If $C(e) < C(e')$ then $e \rightarrow e'$; if $C(e') < C(e)$ then $e' \rightarrow e$; if neither holds then the two events are concurrent. Consider again the events in figure 3.8. From vector time stamps it can be deduced that $a \rightarrow g$, since every element of $[1, 3, 2]$ is greater than or equal to the corresponding element in $[1, 0, 0]$. On the other hand, events b and d are concurrent, since neither $[2, 0, 0] < [1, 1, 0]$ nor $[1, 1, 0] < [2, 0, 0]$.

3.5.3 Limitations of Mattern's System

One limitation of Mattern's presentation of vector-based logical time is that it assumes that a distributed system contains a fixed number, n, of processes. Many distributed systems – such as PVM computations – may be *dynamic*, in the sense that they grow and shrink by creating and destroying processes. Mattern makes no mention of incorporating this dynamism in his system of vector time. Attempting to adapt vector time to describe dynamic distributed systems raises several questions:

- How big should the vector be? If we insist on using vectors of length n and then allow n to vary then things get complicated. Each process will have it's own idea of what n is. If we decide to make the vector include an element for every process that will ever be created, then we must address the resulting efficiency issues.

- How is P_i's clock affected when processes are created or destroyed?
- What should be the initial time vector for a new process?
- Are there performance concerns involved in maintaining vector clocks when hundreds or thousands of processes are created? How do we avoid wasting resources on keeping track of dead processes?

3.6 Extending Vector Time to Dynamically-Sized Systems

Fidge [3] proposed a straightforward extension of Mattern's logical time to allow for dynamic distributed systems.

3.6.1 The Structure of Dynamic Vector Time

As stated before, Mattern's formulation of logical time was a vector of n non-negative integers, where n is the (constant) number of processes in the system and $V[i]$ is the number of events that have occurred in process P_i.

To revise this vector for dynamic systems, we must take several steps. Let $\mathcal{P}$ be the set of legal process id's. [2] The new logical time vector U must be large enough to hold information about all of the processes that are ever created, so let U contain $|\mathcal{P}|$ elements.

[2] These must not be re-used during the course of the distributed computation.

Next we need some representation for the state of each process P_i, the value $U[i]$. In a static system a non-negative integer suffices. But when we allow processes to be created and destroyed a simple integer is not expressive enough.

A process that has not yet been created can be adequately represented by a zero, since zero events have happened in such a process. Still, there is a large qualitative difference between a process that has yet to start (and thus has experienced zero events) and a process that has just started, and has thus experienced exactly one event. This is a large qualitative difference and so a special symbol, $\bot$, is used for a process that has not yet started.

A process that has just started will be denoted by the value one. Hence if P_i has not yet started then $U[i] = \bot$ and if P_i has just started then $U[i] = 1$.

We could choose to represent a terminated process with the index of its last event. But this does not convey all of the information that we know about such a process. We know that it has terminated, and this is a very valuable piece of information that we would like to propagate to other processes. For example, it is much more useful to know that P_7 has terminated than it is to know that 4327 events have transpired in P_7. Consequently we will denote a terminated process with the symbol $\top$.

Thus the possible values for an element of U are $\mathcal{S} = \{\bot, \top, 1, 2, 3, \ldots\}$ and the global logical time vector U must be an element of $\mathcal{S}^{|\mathcal{P}|}$.

Figure 3.9 shows the progress of global logical time in a dynamic system.

Each process P_i in the system will maintain a logical clock, $C_i \in \mathcal{S}^{|\mathcal{P}|}$ that is an approximation to the global logical time U. As before, P_i will derive C_i from time stamps piggybacked on messages from other processes

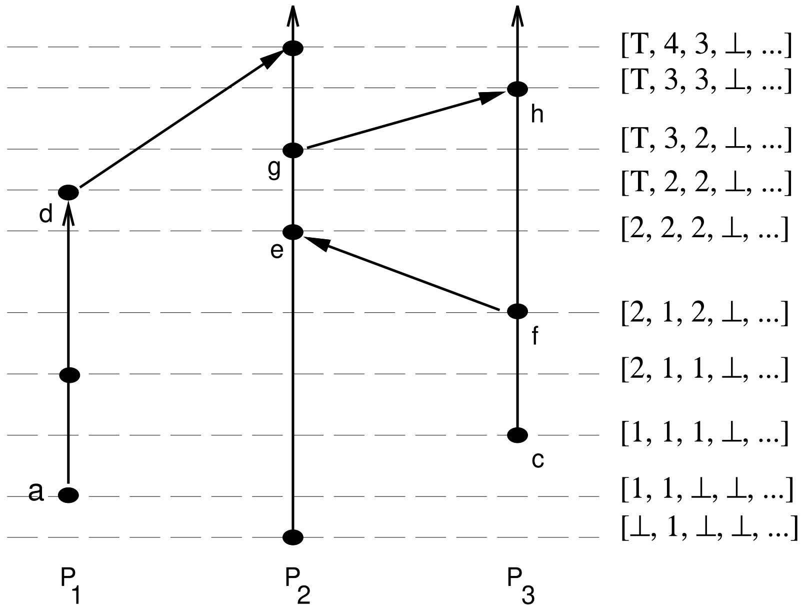

Figure 3.9: Global Dynamic Vector Time

in the system.

3.6.2 Implementation Rules for Dynamic Vector Time

The rules for keeping track of vector time in a dynamic distributed system are similar to the rules for a static system:

1. Before each event in P_i the local clock, C_i, "ticks" by incrementing the local component:

$$C_i[i] \leftarrow C_i[i] + 1$$

2. P_i piggybacks on every outgoing message the current value of C_i.

3. When P_i receives a message with a time-stamp V it updates its approximation of the progress of processes in the system:

$$C_i \leftarrow MAX(C_i, V)$$

   where MAX is the component-wise maximum of its two vector operands using the $\prec$ operator.

4. When a process starts its logical clock is initialized in a manner that depends on the cause of the process's creation:

   (a) If P_i was started by a cause external to the system (e.g., by a user running a program from the terminal), then no other processes in the system have had a causal impact on it. Its clock is initialized as follows:

$$\begin{aligned} C_i &\leftarrow\ <\perp, \perp, \ldots, \perp>; \\ C_i[i] &\leftarrow\ 1 \end{aligned}$$

   (b) If P_i was spawned by another process in the system, P_j, then clearly $e_{spawn} \rightarrow e_{start}$, where e_{spawn} is the event that initiated the process creation, and e_{start} is the event of the execution of the first instruction of P_i. In this case C_i should be initialized as follows:

$$\begin{aligned} C_i &\leftarrow\ t; \\ C_i[i] &\leftarrow\ 1 \end{aligned}$$

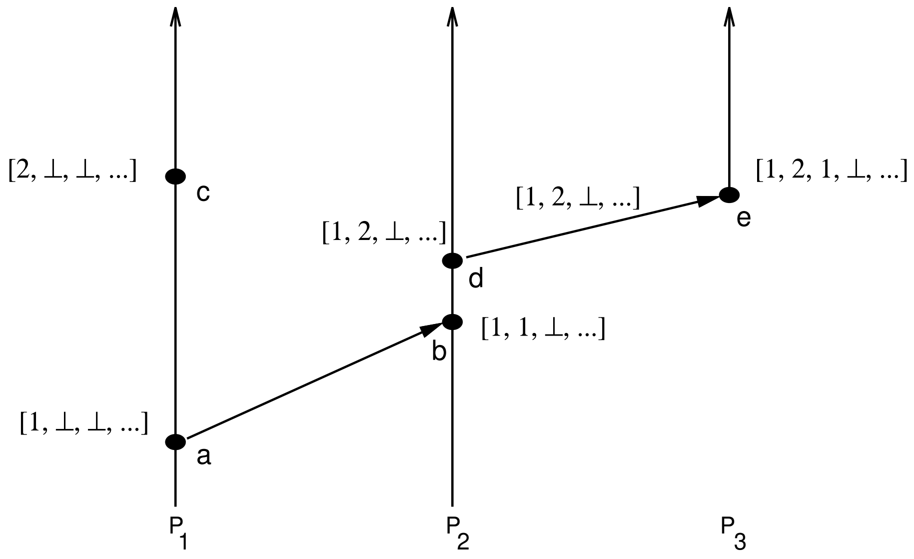

Figure 3.10: Dynamic Vector Time and Causal Process Creation

where t is the value of C_j at the time P_j creates P_i.

5. Just before process P_i terminates it must send a time-stamped message to at least one other live process in the system to let it know that P_i has terminated. If the process terminates unexpectedly, before having a chance to inform another process, the protocol will still work correctly. The remaining live processes will simply be unaware that the task has terminated.

6. When process P_i receives a notification that process P_j has terminated it handles this message as it would any other (by rule 3) and then sets $C_i[j] \leftarrow \top$.

Figure 3.10 illustrates a system where process P_2 spawns process P_3.

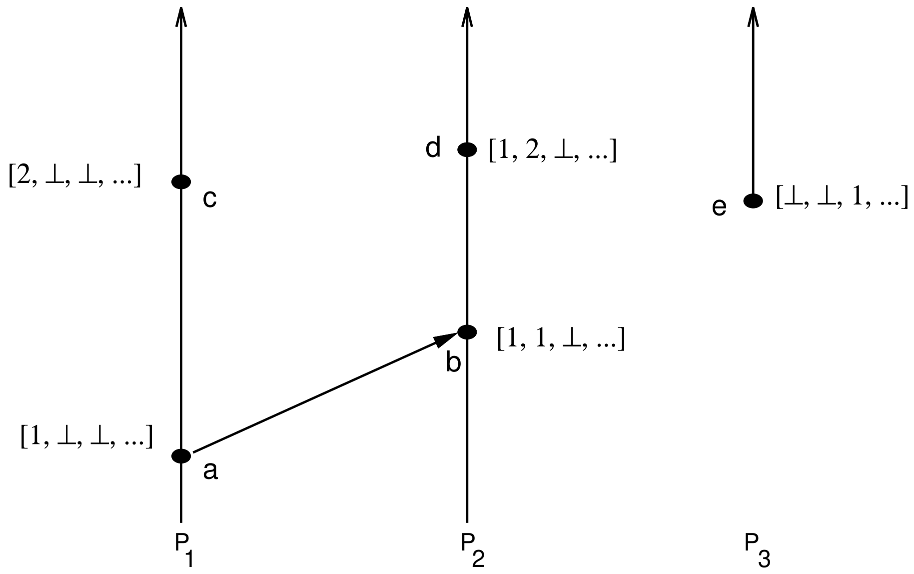

Figure 3.11: Dynamic Vector Time and Independent Process Creation

Figure 3.11 shows the case where the new process, P_3, is started by a cause external to the executing system.

Figure 3.12 illustrates a system where process P_1 terminates and notifies P_2.

3.6.3 Reasoning with Dynamic Vector Time

Mattern's results about the relations between time vectors and the corresponding causal relationships between events can be extended to dynamic distributed systems if we define a total order, "$\prec$", with which we can compare process states in a dynamic system. Let $\perp \prec 1 \prec 2 \prec 3 \prec \cdots \prec \top$, or, more formally:

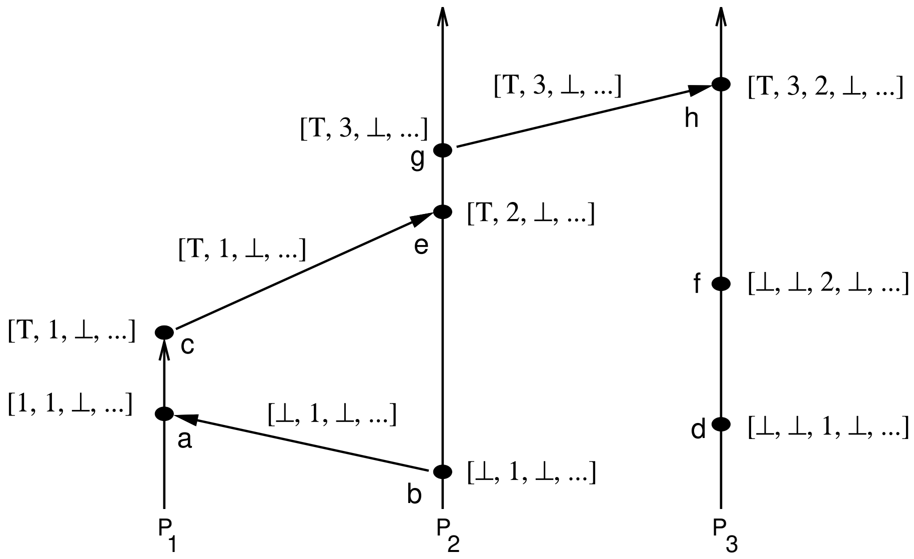

Figure 3.12: Dynamic Vector Time and Process Termination

Definition 3.5. *Let $\prec$ be the binary relation on $\mathcal{S} \times \mathcal{S}$ given by*

$$\prec \; = \; < \; \cup \{(\perp, x) \mid x \in \mathcal{S} - \{\perp\}\} \; \cup \; \{(x, \top) \mid x \in \mathcal{S} - \{\top\}\}$$

where $<$ is the standard less-than relation on the non-negative integers.

Using $\prec$ it is a trivial step to define $\preceq$:

Definition 3.6. *Let $\preceq$ be a binary relation on $\mathcal{S} \times \mathcal{S}$. For any $x, y \in \mathcal{S}$,*

$$x \preceq y \iff (x \prec y) \vee (x = y)$$

Now we can define relations on time vectors in $\mathcal{S}^{|\mathcal{P}|}$, that is, logical time vectors for a dynamic distributed system:

Definition 3.7. *For two time vectors $U, V \in \mathcal{S}^{|\mathcal{P}|}$*

- $U \leq V \iff \forall i :\ U[i] \preceq V[i]$
- $U < V \iff U \leq V \ \wedge\ U \neq V$
- $U \mid V \iff \neg(U < V) \ \wedge\ \neg(V < U)$

This result will be useful later:

Theorem 3.8. *The relation $< \subseteq \mathcal{S}^{|\mathcal{P}|} \times \mathcal{S}^{|\mathcal{P}|}$ is transitive.*

Proof. Suppose $U < V$ and $V < W$. Then $U \leq V$, $U \neq V$, $V \leq W$, and $V \neq W$.

Since $U \leq V$, U is less than or equal to V on a component-wise basis; likewise since $V \leq W$, V is less than or equal to W on a component-wise basis. Consequently, $U \leq W$.

Since $U \leq V$ and $U \neq V$, there must be at least one component of V that is greater than the corresponding component of U. Likewise there must be at least one component of W that is greater than the corresponding component of V. Thus there is at least one component of W that is greater than the corresponding component in U, and $U \neq W$.

Since $U \leq W$ and $U \neq W$ we can conclude $U < W$.

□

Just as in Mattern's original formulation, the $<$ and $\mid$ relations are isomorphic to the causal relationships between events:

Theorem 3.9. $\forall\, e, e' \in \mathcal{E} : \; e \to e' \iff C(e) < C(e')$

Proof. To prove this by induction we will need an order in which to consider the events. Recalling that $\mathcal{E}$ is the set of events in a distributed system, let $\hookrightarrow \subseteq \mathcal{E} \times \mathcal{E}$ be any total order of these events which respects causality. That is, $\hookrightarrow$ is any total order such that $\hookrightarrow \supseteq \to$. Let e_i denote the i-th event in $\mathcal{E}$, as ordered by $\hookrightarrow$.

We can prove this theorem by induction on the events in $\mathcal{E}$, as ordered by $\hookrightarrow$. Let P represent the proposition $e \to e' \iff C(e) < C(e')$.

P holds for e_1, the first event in $\mathcal{E}$ as ordered by $\hookrightarrow$, since $e_1 \not\to e_1$ and $C(e_1) \not< C(e_1)$.

Suppose that P holds for all pairs of events in $\mathcal{E}$ up to but not including e_n, the n–th event as ordered by $\hookrightarrow$. We must then show that P is true for all pairs of events up to and including e_n.

- First we show that for all e_i such that $i \leq n$, $e_i \to e_n \Rightarrow C(e_i) < C(e_n)$. Suppose $e_i \to e_n$. Then either e_i is a direct cause of e_n or e_i is an indirect cause. If e_i is a *direct* cause of e_n then either
  - e_i is the immediately-preceding event in the same process as e_n. Then $C(e_n)$ is derived from $C(e_i)$ by incrementing one of the vector elements (rule 1), so $C(e_i) < C(e_n)$.
  - e_i is a message send and e_n is the corresponding receive. Then by rules 2 and 3, $C(e_n)$ is calculated by taking the maximum of $C(e_i)$ and another time vector, and then incrementing one of its elements. Thus $C(e_i) < C(e_n)$.

  - e_i is the sending of a notification message for the termination of process P_j, and e_n is the receipt of this notification message in P_k. Then by rules 5 and 6 this is handled as a regular receipt, and then $C_k[j] \leftarrow \top$. By reasoning identical to the step above, and because $\top$ is the greatest value in the $\prec$ relation, $C(e_i) < C(e_n)$.
  - e_i is a request to create a process and e_n is the start of that process. Then according to rule 4, just as in the case of a message send, $C(e_i) < C(e_n)$.

  If e_i is an *indirect* cause of e_n then there exists some *direct* cause e_j such that $e_i \rightarrow e_j$ and $e_j \rightarrow e_n$. Since $e_j \rightarrow e_n$ it is also the case that $e_j \hookrightarrow e_n$. Thus by the inductive hypothesis we know $C(e_i) < C(e_j)$. Since e_j is a direct cause of e_n, by reasoning identical to the above step, we know that $C(e_j) < C(e_n)$. Thus by the transitivity of $<$, $C(e_i) < C(e_n)$.

- Next we show that for all e_i such that $i \leq n$, $C(e_i) < C(e_n) \Rightarrow e_i \rightarrow e_n$. Suppose $C(e_i) < C(e_n)$. Now either e_i and e_n are in the same process or they aren't.

  Suppose e_i and e_n are in the same process. Since e_i and e_n are in the same process they cannot be concurrent; one must happen-before the other. Thus the fact the $e_i \hookrightarrow e_j$ means that $e_i \rightarrow e_n$.

  Suppose e_i and e_n are in different processes, P_j and P_k (see figure 3.13). By definition, $C(e_i) < C(e_n)$ implies that $C(e_i)[j] \leq C(e_n)[j]$. But given the implementation rules, there are only a few ways that

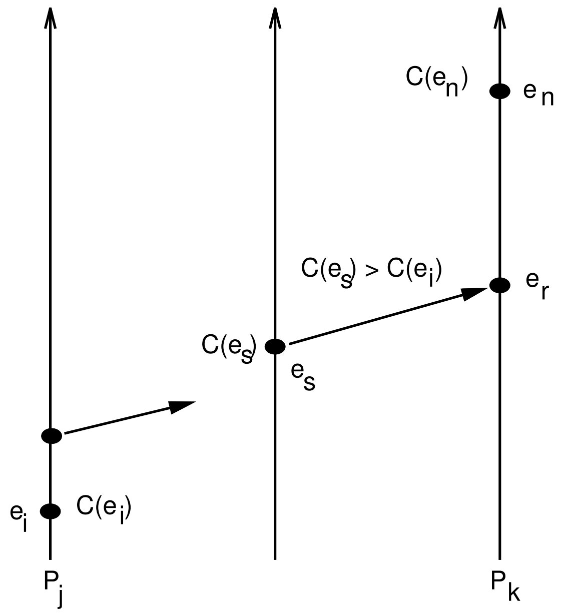

Figure 3.13: Time Stamp Order Implies Causal Order

P_k's clock could attain this value:

– Some process sends a message to P_k with a time stamp whose value included $C(e_n)[j]$ events from P_j. Let e_s be the sending of this message and e_r be the receipt of this message, which must occur before e_n. Since $C(e_s)[j] = C(e_n)[j] \geq C(e_i)[j]$, $C(e_s)$ must also incorporate all of the other elements of $C(e_i)$. Thus $C(e_i) < C(e_s)$ and by the inductive hypothesis, $e_i \rightarrow e_s$. By definition $e_s \rightarrow e_r$ and as mentioned, $e_r \rightarrow e_n$. Thus $e_i \rightarrow e_n$.

– Some process sends a termination notification message to P_k with a time stamp whose value included $C(e_n)[j]$ events from P_j. The reasoning is as above.

- P_k was spawned by some event e_s in the system. The reasoning is similar. The first event in P_k, e_r, must occur before e_n. Since $C(e_s)[j] = C(e_n)[j] \geq C(e_i)[j]$, $C(e_s)$ must also incorporate all of the other elements of $C(e_i)$. Thus $C(e_i) < C(e_s)$ and by the inductive hypothesis, $e_i \rightarrow e_s$. But $e_s \rightarrow e_r$ and as mentioned, $e_r \rightarrow e_n$. Thus $e_i \rightarrow e_n$.

$\square$

Just as causality ($\rightarrow$) is isomorphic to the $<$ relation, so is concurrency ($\|$) isomorphic to the $|$ relation:

Theorem 3.10. $\forall\, e, e' \in \mathcal{E} : \; e \,\|\, e' \iff C(e) \,|\, C(e')$

Proof. Suppose $e \,\|\, e'$. Then $e \not\rightarrow e'$ and $e' \not\rightarrow e$. Thus by theorem 3.9, $\neg(C(e) < C(e'))$ and $\neg(C(e') < C(e))$. So by the definition of $|$, $C(e) \,|\, C(e')$.

Suppose $C(e) \,|\, C(e')$. Then $\neg(C(e) < C(e'))$ and $\neg(C(e') < C(e))$. Thus $e \not\rightarrow e'$ and $e' \not\rightarrow e$, which means that $e \,\|\, e'$ $\square$

With these theorems in hand, all we have to do to determine the causal relationship between two events e and e' in a *dynamic* distributed system is to check whether $C(e) < C(e')$ or $C(e') < C(e)$. If $C(e) < C(e')$ then $e \rightarrow e'$; if $C(e') < C(e)$ then $e' \rightarrow e$; if neither holds then the two events are concurrent.

Chapter 4

Implementation of Dynamic Vector Time for PVM

This chapter describes the implementation of the protocol specified in the previous chapter in the context of PVM version 3.3.

4.1 The PVM Vector Clock

Every task and PVMD in a PVM virtual machine maintains a vector clock data structure that represents the process's current approximation to the system's global dynamic vector time. This clock is stored as pairs of task id's and clock values hashed by task id. A $\perp$ value for process P_i is implicit in the absence of a $(i, clockvalue)$ pair in the hash table. A $\top$ value for P_i is represented by the pair $(i,$ `MAXINT`$)$, where `MAXINT` is the maximum possible positive integer on the given platform (usually `7FFFFFFF` hexadecimal or 2147483647 decimal).

4.2 Changes to the PVM API

Adding vector time to PVM necessitated the introduction of two new calls to the PVM library interface. A third arose as a convenient utility function in the implementation of the others.

- The first, `pvm_event`, increments by one the local component of a task's vector clock:

  ```
  int info = pvm_event();
  ```

  The error code `PvmClockOverflow` will be returned by `pvm_event` if the local component of the clock overflows to a negative value.

- The second call, `pvm_clock`, returns a pointer to a copy of the local process's vector clock:

  ```
  struct pvmclockinfo {
     int ci_tid;   /* task id */
     int ci_val;   /* latest clock value from this task */
  };

  struct pvmclockinfo *clockptr;
  int ntasks;

  int info = pvm_clock(&ntasks, &clockptr);
  ```

  This copy of the local vector clock is stored as an array of `ci_tid`, `ci_val` pairs. This will occupy $2 \cdot sizoef(int) \cdot ntasks$ bytes, where *ntasks* is the number of tasks in the entire history of the virtual machine, which may run into the thousands. More efficient implementations are possible, but at a significant cost in simplicity. In any case, at every invocation of `pvm_clock` the last image of the vector clock is deallocated. This is identical to the manner in which PVM handles arrays containing information about the configuration of the PVM virtual machine.

- A third routine, `pvm_seek` was added because of the need for a general mechanism to read the time stamp piggybacked on each message.

  This turned out to be a non-trivial task because of two facts. First, PVM encodes all task-to-task messages using XDR. Second, existing facilities in PVM allowed only sequential access to the contents of a newly-arrived buffer. That is, the program must first "unpack" the data at the beginning of the message before the data at the end of the message can be accessed. This is implemented with a "read pointer" that scans along the length of the message. When data is unpacked, the data at the current location of the pointer is returned and the pointer is advanced to the next available data in the buffer. The `pvm_seek` function was added to advance the pointer for the active receive buffer to a given location without having to read the intervening data:

  ```
  int info = pvm_seek(int pos);
  ```

  This function can only advance the cursor; it cannot retract the cursor.

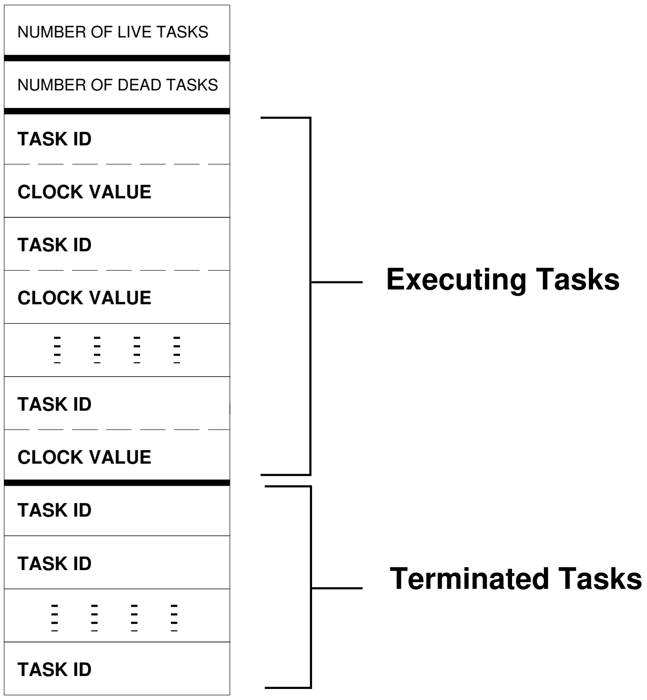

Figure 4.1: Time Stamp Format for PVM Messages

4.3 Piggybacking Time Stamps

Vector time stamps are added automatically to the end of every message sent by a PVM task or PVMD, except messages from a task or PVMD to itself. All messages have a new value, `tstamp`, in their header that records the offset into the message contents of the time stamp data. Figure 4.1 shows the format of time stamps that are placed at the end of messages.

The function `pack_time_stamp` adds a time stamp to every message leaving a PVM task, and the function `pluck_off_time_stamp` reads this time stamp, incorporates its clock values into the local vector clock, and releases the memory occupied by the time stamp. The time stamp is removed before PVM library clients can read the message. Thus no PVM task can ever discover that a message it receives had a time stamp.

The function `pack_pvmd_time_stamp` adds a time stamp to all messages that PVMD's send. Its counterpart `pluck_off_pvmd_time_stamp` reads and discards the time stamps on messages bound for PVMD's.

Piggybacking and removing time stamps from PVM messages was a not as easy as one would have hoped, for PVM uses a doubly-linked list of fixed-size message buffers to store messages. In addition, all time stamp values had to be placed in a standard XDR-encoding so that they would be readable by different platforms.

By default all task-to-task messages in a PVM computation are routed one message buffer at a time through the local PVMD. Unfortunately the PVMD, when forwarding task-to-task messages, does not assemble these pieces into a complete message. Consequently it is very difficult to find the time stamp data in an incoming message. Modifying the PVMD so that forwarded messages are assembled would have required a major re-write of the architecture of message-passing in the PVMD, and as a consequence this feature was not implemented. As a result, PVMD's do not extract time stamps from task-to-task messages that they are forwarding; only from messages that they assemble. Likewise, a PVMD only adds time stamps to messages that it initiates; *not* to messages that it is forwarding.

As a consequence of this fact PVMD's only interact with the vector time system insofar as they send and receive messages. They have no effect on vector time stamps when forwarding messages. This is a loss of some information about the causal interactions in the system. However, this is just as it would be if messages were sent directly between PVM tasks. Since this is the abstraction offered by PVM, this is no great loss.

In fact, one could make an argument that PVMD's should not interact with the vector time scheme at all. After all, this interaction introduces unintended causal relationships that the programmer is unlikely to understand. On the other hand, since PVMD's keep track of vector time they naturally pass on their clocks to newly-spawned tasks. In addition, this design allows the programmer to see a more detailed view of the causal structure of the system.

4.4 Task Creation and Termination

The dynamic vector time scheme requires special attention the creation and termination of processes (rule 4). PVM complies with this rule in a straightforward manner.

When a PVM task or PVMD with task id i is created it initializes its clock hash table to empty, except for a single pair $(i, 1)$. This correctly handles the case where the process's creation was not causally affected by any task in the distributed computation.

However, this initialization is rarely the last word. In order for a newly-created PVM task to enroll in the PVM virtual machine it must send a message to its local PVMD and receive a reply message. This reply message will contain the vector time clock of the local PVMD. Likewise, after a new PVMD is created it receives some messages from the master PVMD. These messages will contain the master PVMD's clock value. In both cases, a new process in the virtual machine will initialize its clock to a single entry but almost immediately fill its clock with the clock of the process responsible for its creation. The only process that will not immediately receive a clock value is the master PVMD, the first PVM process created.

4.5 Task ID's

The dynamic vector time scheme requires that process id's not be reused. In a dynamic system which grows rapidly, process id's as assigned by the operating system or message-passing library may, in theory, be recycled. This would invalidate our vector time scheme. Thus if the operating system or message-passing library that the application is employing recycles process id's, an extra layer of software must be inserted to map synthetic process id's employed in the logical time scheme to the actual operating system or message-passing environment id's.

For PVM this is not a practical concern, since task id's are ordered pairs including a *hostpart* and a *localpart*, where *hostpart* is represented in $2^{12}-1$ bits and *localpart* is represented in $2^{18}-1$ bits. This means that there can be up to 262,143 tasks on a single host in the course of a computation without having to reuse id's. This is sufficient for all but the oddest and longest-lived computations.

4.6 Caching

4.6.1 Motivation

As originally implemented, the time stamp protocol sent the entire clock with every message: first a list of $(taskid, clockvlue)$ pairs for all of the live processes, then a list of the task id's of the dead processes. But several researchers (for example, Fidge [3]) have noted that this is unnecessary. Since PVM tasks and PVMD's will typically communicate quite frequently, space in each message is dedicated to clock values that haven't changed since the last message. If P_i caches the latest clock values that it has sent to and received from P_j then it need only send the values that have changed since the cache entry was last modified.

If we use both cache-on-send and cache-on-receive then the cache approximates the contents of other processes' clocks. The cache contains values that we know it would be senseless to send out to another process, because 1) we already sent the value, or 2) we already received the value.

For a P_i engaged in sending messages to many other processes this cache could potentially consume a large amount of memory. However, a fixed-size cache can serve as a compromise between the small message size afforded by this technique and the attendant cost of maintaining this information in memory.

4.6.2 Implementation

The cache keeps a record of the c latest clock values that have been communicated with some set of processes, $\{P_l, P_m, \ldots, P_n\}$. This set of cached clocks varies dynamically over the course of execution so as to maintain the most-recently used entries. Whenever an entry in the cache must be replaced by a new entry, the least recently used cache entry is discarded.

The central new routine of this scheme is the function

```
check_cache_for_send(int tid)
```

Given the *tid* of a task that we are about to send a time stamp to, this routine:

1. Looks in the cache for an entry containing the latest clock values that we communicated with the task with the give *tid*.

2. Whether it has a hit or not, constructs set of clock values that must be sent to task *tid* to keep it up to date.

3. Ensures that the cache contains an entry with the latest clock values that have been communicated with this task. This entry is moved to the head of the cache. If the cache grows beyond a certain size, the tail-most (least-recently-used) entry is destroyed.

4. Returns the set of clock values that must be sent to task *tid*.

The sending task then packs these clock values onto the tail of the message.

4.6.3 Evaluation

It is clear that the time stamp cache is very effective for systems with a small number of processes. If the cache holds c entries and a process communicates only with m other processes, $m \leq c$, then there will be a cache hit on every send. Since there are m processes involved, at most $m - 1$ clock values will have changed since the last communication with a given process. Thus the overhead for a time stamp is at most $2 + 2 \cdot (m - 1)$ integers. On a system with $c = m = 10$, this will amount to a 80 byte overhead for a time stamp, which is acceptable.

How common is it for a process to have such a limited communication pattern? Quite common. Many paradigms in parallel and distributed computing involve a process communicating with a few others: with its 2 neighbors in a pipeline or ring, a single central server process, or one parent and several children in a tree configuration.

The biggest advantage offered by a cache is that very few $\top$ values are sent in time stamps, since these values will never change from the cached value. Consequently, the dynamic vector time system is sending around only time stamp values for live, executing processes. In other words, a cache gives us dynamic vector time for the same cost as static vector time.

Unfortunately, a cache will offer little benefit in a system where processes often communicate with many other processes. One scenario that will soundly defeat the cache is a series of message broadcasts to a number of processes larger than c. In this situation the broadcasting task is never able to use the cache, and the broadcast messages must each carry a full time stamp. Similarly, because of the least-recently-used replacement policy, if a process cyclically communicates with $c + 1$ or more tasks, each send will find that the pertinent cache entry has recently been discarded. Again, this will force every single outgoing message to have a full-sized time stamp.

Chapter 5

The Causally-Structured Output Stream Facility

This chapter presents a definition of causally-structured output streams and discusses the implementation of this construct for PVM.

5.1 Definition and Structure

Sequential output streams were defined as a set of entries together with the $\Rightarrow$ total order defined over those entries. A *causally-structured output stream* can be defined similarly:

Definition 5.1. *A causally-structured output stream is a tuple* $(R, \mathcal{E}, \rightarrow)$ *where* $R = \{r_1, r_2, \ldots r_n\}$ *is a set of entries, and* $\rightarrow \subseteq \mathcal{E} \times \mathcal{E}$ *is the happened-before relation that relates the events* $\mathcal{E} = \{e_1, e_2, \ldots e_n\}$ *that are directly responsible for the creation of these entries.*

The $\rightarrow$ relation orders both the set of events and the set of entries according to causality. The entries in the stream can be from any process on any host in the distributed system. And just as with sequential streams, the entries themselves can be any data. Also, since $\rightarrow$ is a partial order, there will likely be entries in the stream which are concurrent.

For example, consider figure 5.1, which depicts a simple output stream:

$$(\{r_1, r_2, r_3, r_4\}, \{e_1, e_2, e_3, e_4\}, \{(e_1, e_2), (e_1, e_3), (e_1, e_4), (e_2, e_3)\})$$

In this stream event e_1 writes out the string `"a"`, e_2 writes `"b"`, e_3 writes `"c"`, and event e_4 writes `"d"`. In this stream `"a"` is before `"b"` which is before `"c"`. However, `"d"` is only after `"a"`; it is not related to any other stream entry.

The definition in terms of a set of events and a partial order leaves to applications and tools the presentation of the contents of the output stream. In particular, this allows for viewing an output stream as a directed acyclic graph similar to figure 5.1, as well as a flattened text block.

Causally-structured streams depend, in the abstract, on a causal relation between their elements. Since the relationships between vector time stamps are isomorphic to the causal happened-before relation, $\rightarrow$, a vector time stamp system like that developed in earlier chapters will suffice to provide the partial order for the stream.

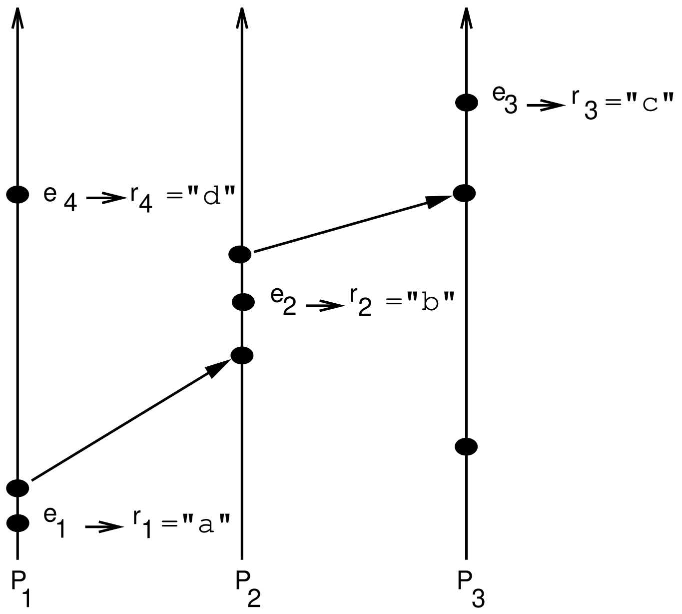

Figure 5.1: A Causally-Structured Stream

5.2 Design Considerations

The PVM causally-structured output stream facility was designed with the following considerations in mind:

- The programming interface should resemble a standard sequential output stream as much as possible. It should be as simple as possible.

- The implementation of causally-structured output streams should not introduce any extra causal relationships into the system. This means that they will have to be implemented without message-passing, or at least without message passing *inside* the system *during* the computation. Clearly some message-passing is required to move the stream entries from the host on which they initially reside to the host on which they are viewed.

- The implementation should perturb the timing of the system – as measured by real-time clocks – as little as possible. Interfering with the timing of a distributed system risks the occurrence of a "Heisenbug," so named because it is an error that arises due to timing changes caused by instrumentation of the distributed system for debugging, monitoring, or visualization [1].

- The implementation should be able to handle large output streams. This means that it will have to use files for storing the stream entries, rather than relying solely on memory.

- The implementation should allow the programmer easy and flexible viewing of the components of the stream. This means that, as much as possible, the files comprising the output stream should be human-readable – in case the programmer wants to manually inspect them – and easily machine-parsable – in case the programmer wants to build tools to view them.

- The implementation should allow each process to write to multiple output streams. Just as with sequential output streams, this allows the programmer to direct output to a stream according to its semantics. This will let tools provide a flexible view of the system by allowing the programmer to interactively specify a set of streams to view.

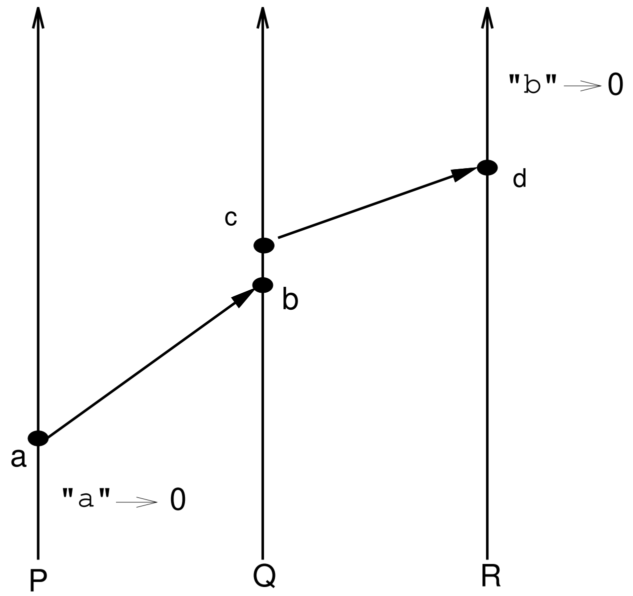

Figure 5.2: Ordering with Closed Streams

5.3 Interaction Between Output Streams

The fact that each process in a system can open any number of output streams raises several questions.

First, what happens when some processes in a system open stream k and some do not? Consider figure 5.2, where process P and R open and write to stream 0, whereas process Q does not. In this scenario, P sends a message to Q after writing `"a"` to stream 0. After Q receives this message it sends a message to R, which then prints out `"b"` on stream 0. What is the relationship between `"a"` and `"b"`?

Because vector time is implemented independently from output streams in PVM, event a precedes event d. Thus `"a"` and `"b"` are in this order in

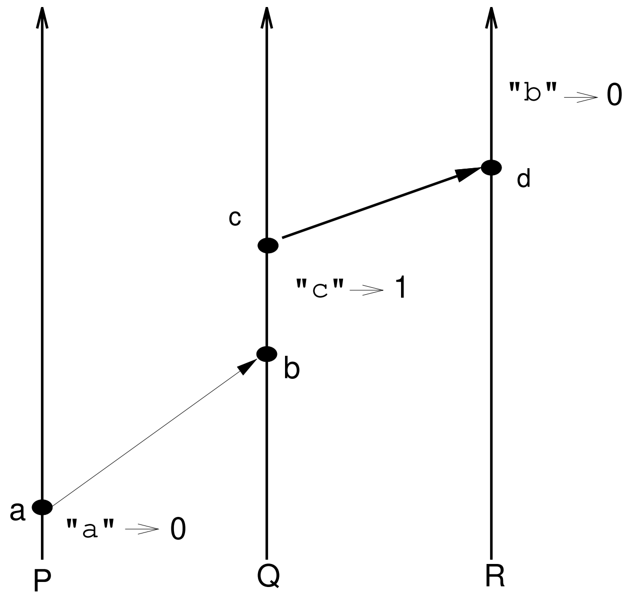

Figure 5.3: Ordering with Multiple Streams

the output stream.

The other interesting case involves communicating processes writing to different streams. Consider figure 5.3. In this scenario processes P and R again write to stream 0, but process Q this time writes the string `"c"` to a *different* stream, stream 1. Again, since vector time is maintained independently of the output streams, `"a"` precedes `"b"`. Furthermore, if one were to consider the union of streams 0 and 1, each of `"a"`, `"c"`, and `"b"` would be a member of this stream, in that order.

5.4 The API

The programming interface for a causally-structured output stream in PVM involves three simple calls:

```
int n;
int info;

info = pvm_sopen(n);
info = pvm_sprintf(n, "format string", ...);
info = pvm_sclose(n);
```

Each output stream in the PVM virtual machine has a numeric identifier, appearing above as `n`. The standard output stream for the PVM virtual machine is stream number 0 and the standard error output stream is stream number 1. The PVM interfaces include the predefined constants `PVM_STDOUT` and `PVM_STDERR` to denote these streams. Though streams 0 and 1 are used for these purposes by convention, any non-negative number may be used.

A task must call `pvm_sopen` to request that a connection be opened to a given output stream before it may write messages to this stream. If the given stream is already open, the `PvmStreamOpened` error code is returned. Each task may open as many output streams as it wishes.

While a stream is open a process may write to it by calling `pvm_sprintf`, providing a destination stream and a variable-length argument list in the style of the C `printf` standard library function. If the given stream has not opened for writing then the error `PvmStreamNotOpened` is returned. If the given stream was already opened and closed then the `PvmStreamClosed` code is returned.

After the task is done writing to a stream it should close its connection to the stream by calling `pvm_sclose`. If the given stream was not open for writing in the first place then the `PvmStreamNotOpened` error code is returned. If the given stream was already closed then the `PvmStreamClosed` code is returned. Once a connection to a given stream has been closed, it may not be re-opened. This ensures that a stream-monitoring tool can know that the process is really done writing to that stream.

5.5 Implementation

The output of `pvm_sprintf` routines is written to a file on the host on which a task is running. This file is named:

```
/tmp/lpvmstream.uUID.tTTTT.sSSS
```

Here `UID` is the user id of the user running the task, `TTTT` is the task id of the task, and `SSSS` is the id of the stream.

When a process opens a connection to a stream, lpvm creates a trace file of the appropriate name in the `/tmp` directory of the local host and opens it for writing with the C standard input/output library.

The `pvm_sprintf` function then writes vector time-stamped messages to this file using the C stdio library. Because this function uses buffered output to a local disk, the timing perturbation effect of generating the output stream entries is negligible.

When `pvm_sclose` closes the connection to the stream, it closes the C `FILE` associated with the connection and renames the file

`/tmp/lpvmstream.uUID.tTTTT.sSSS.done`

This ensures that no other process will accidentally read the file when it is unfinished and potentially in an inconsistent state. This also allows a program to see which files on a host have been closed with a simple directory listing. An alternative to renaming the file would be to use kernel or library file locking facilities to let the viewer know that the task is still writing to the file.

Note that these files in the `/tmp` directories of all of the hosts do not, in and of themselves, constitute a causally-structured stream. The causally-structured output stream is an abstraction whose components *include* the entries stored in these files. But the other important part of the output stream is the $\rightarrow$ relation defined over these entries. This relation can be mechanically derived from the time stamps on the stream entries. Thus the information needed to construct the streams is contained entirely in these trace files, along with the names of these files. But the causally-structured output stream is itself merely an abstraction guiding the interpretation of this data.

Chapter 6

The Stream Viewing Tool

6.1 Goals

The stream-viewing application SV was constructed with the following goals in mind:

- To provide an example of a useful tool that uses causally-structured output streams to view the output of a distributed computation.
- To test the design and implementation of the PVM causally-structured output stream facility.
- To provide an easy-to-use, clear, and informative graphical display of the output stream, its entries, and their causal relationships. The tool should avoid the "angry fruit salad on drugs" problem mentioned earlier.
- To provide a case study involving the use of causally-structured output streams to debug a program.

6.2 User Interface Issues

The central focus of the viewing tool is a set of scrollable, resizable windows displaying the union of one or more streams. The user can create a new viewing window and change the set of streams that is being displayed in a given window.

The stream-viewing tool allows two views of a set of streams:

Flat Textual A random list of the stream entries, in an order *consistent with causality.* The user can have the tool reorder the trace outputs into another random order consistent with causality. This allows a programmer to quickly see several of the possible interleavings of concurrent events without having to re-execute the application. See figure 6.1.

Space-Time A view inspired by the traditional Lamport space-time diagram [7] of a distributed system. Entry nodes display the contents of the entry and arrows between nodes denote causal relationships between entries. All direct causal relationships will result in an arrow, while those causal relationships that are derivable from direct causal relationships will be left out of the diagram to reduce clutter. It is easy enough for the programmer to visually find the transitive causal relations. See figure 6.2.

In both views the programmer can click on a stream entry to select it and specify an action to apply to the entry:

- Highlight the given entry and paint all the stream entries concurrent

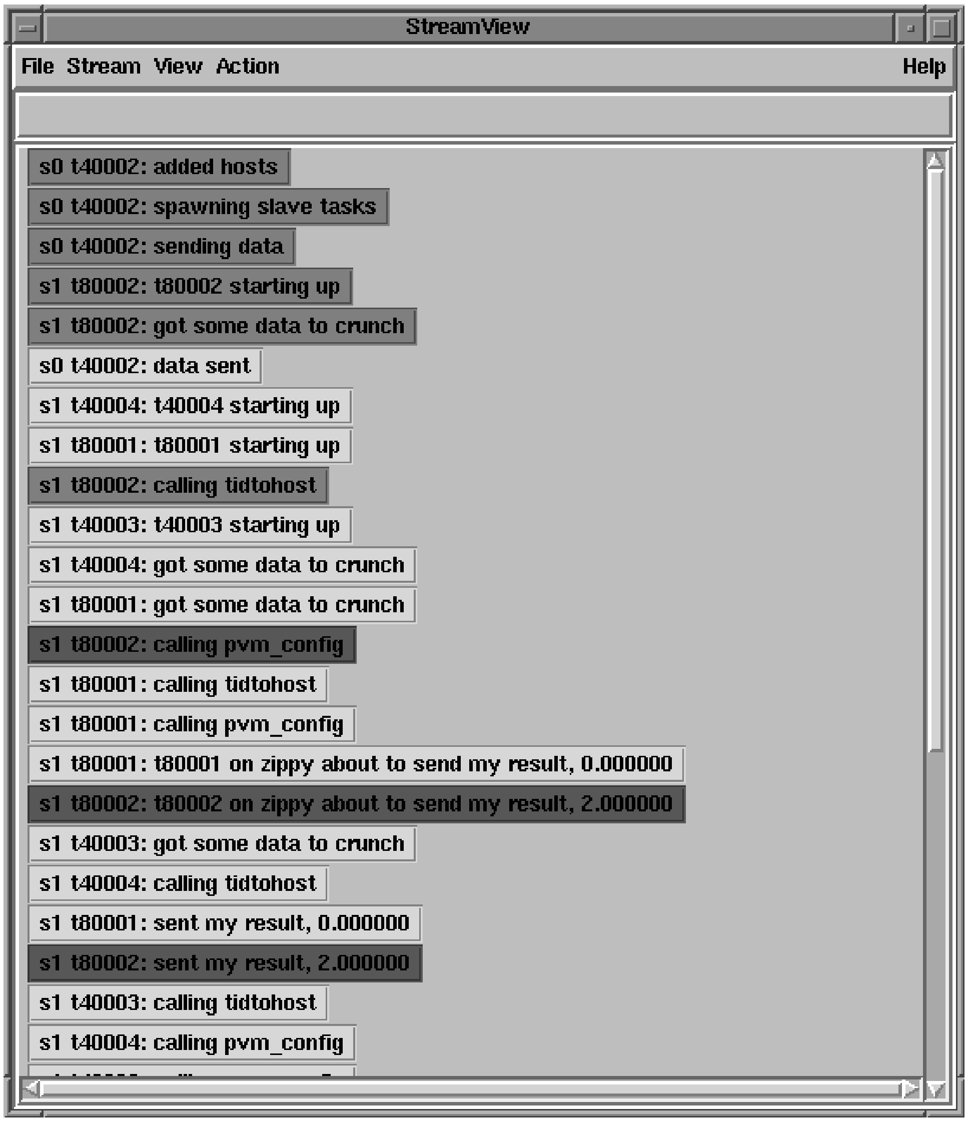

Figure 6.1: A Flat Textual View

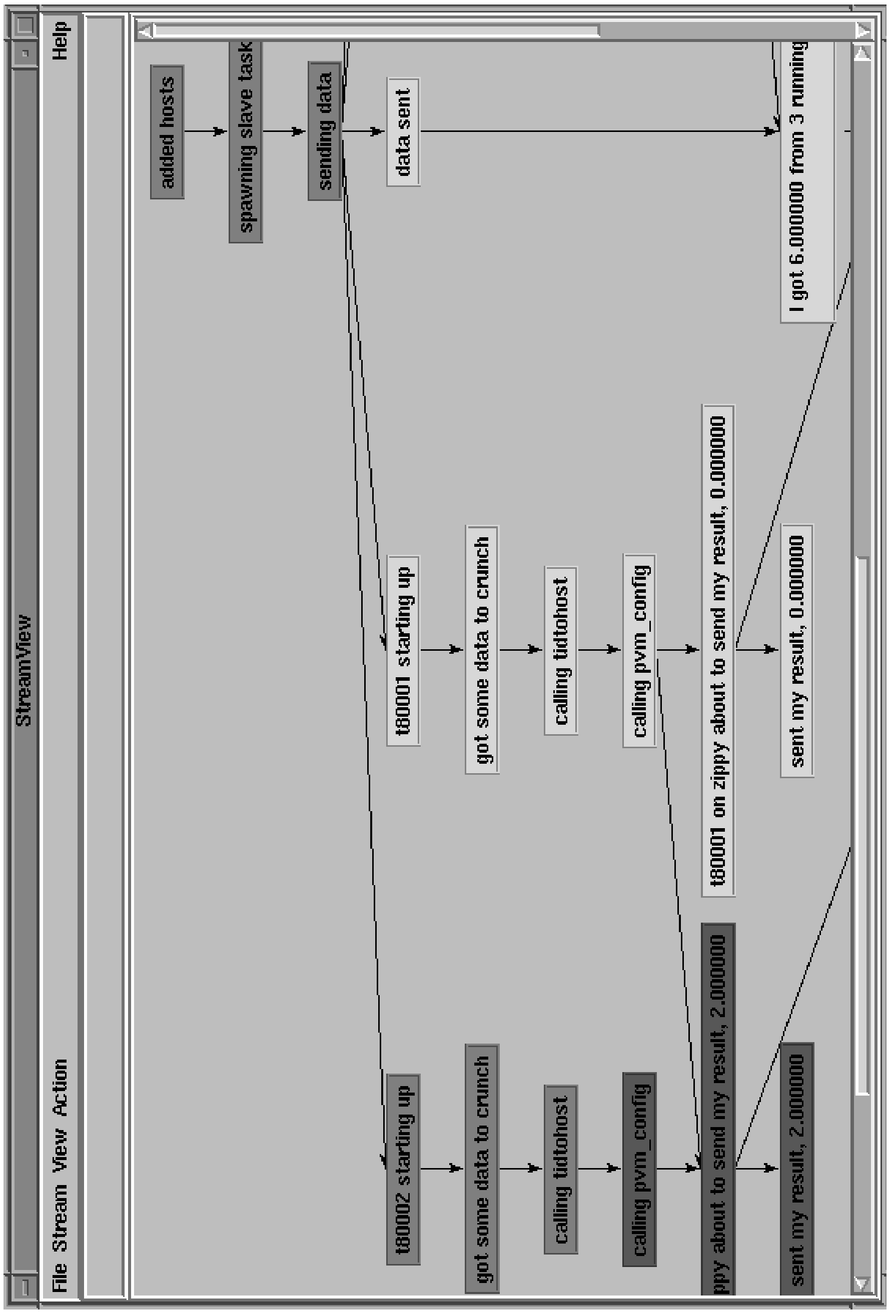

Figure 6.2: A Space-Time View

to, causally preceding, or causally following this one.

- Highlight the given entry and paint all entries in the stream such that they are color-coded by their causal relation to this entry (green = preceding, yellow = concurrent, red = following).

- Create a window containing detailed information about a stream entry, including the stream number, source task id, entry contents, and time stamp.

6.3 Implementation Issues

SV is a PVM process enrolled in the PVM virtual machine. It uses PVM to obtain a list of the hosts that are currently in the PVM virtual machine. This does introduce SV into the causal relation, but not in a significant way since all monitoring is performed post-mortem. An alternative would be to have command line and interactive facilities to specify the set of hosts on which SV looks for trace files.

SV keeps a database of stream elements printed to the stream files in the system that have been closed. Currently this database is kept entirely in memory. In order to support large output streams SV could keep this database on the local disk with an in-memory cache.

SV has an "update" command that updates the given streams in the database by scanning all of the closed stream files from the `/tmp` directories of the hosts in the PVM virtual machine. It integrates into the database the contents of any of these files that have been finished since the last such operation. Because the SV user can specify explicitly when – after the termination of the processes – this update is made, the monitoring of the distributed system has no causal impact on the system.

The database of trace outputs and the corresponding view are static and will *not* be updated as the PVM computation proceeds. Real-time updates would require notification of the viewing tool via message-passing, which would impact the timing and causal structure of the computation. Instead, if the programmer wants to update the display s/he can explicitly request (by selecting the "update" command) that the latest available stream output be displayed.

The SV user can clear the contents of the currently-displayed streams so that output from multiple executions of a program can be viewed in the same SV session.

6.4 Implementation Languages

The stream-viewing tool is implemented in C++ and Tcl/Tk [11]. The data model – all of the computationally-intensive manipulations of stream entries and their text – is in C++ in order to maximize performance. The entire user interface is written in Tcl/Tk in order to provide a standard, easy-to-use user interface for the tool.

Rather than extending Tcl/Tk via static linkage, the C++ back end interacts with WISH via two pipes. One pipe carries commands from the back end to the Tcl/Tk front end. The other carries information from the Tcl/Tk interpreter to the back end.

6.5 Design

The C++ back-end of the stream-viewing tool has a simple object-oriented design with the following classes:

`Window` – A stream-viewing window, with a constituent `StreamView`, represented on the X display by a menu bar, a Tk canvas widget for drawing, and two scroll bars. One of the attributes of a `Window` is the set of streams that it is displaying. Another attribute is the type of representation of the set of streams – flat textual or space-time.

`StreamView` – A display of a set of PVM streams. This is an abstract class whose subclasses are `FlatStreamView` and `SpaceTimeStreamView`. These subclasses handle the drawing of a representation for these streams.

`Events` – Encapsulates all the information that the stream viewer knows about the stream output entries from PVM task trace files that have been closed. Includes an explicit representation of the $\rightarrow$ relation. This is currently represented as an array. Though this will not suffice for large systems, it is easily extendible by changing the implementation of the class to use a more efficient data structure.

`Event` – Consists of the text from a `pvm_sprintf` call, the stream to which the entry belongs, the task id of the source task, and a time stamp indicating the source and time of the call.

Chapter 7

A Case Study

This chapter presents a case study in the use of dynamic vector time, causally-structured output streams, and the stream viewing tool to debug a distributed application. There are other uses for these systems, but debugging is the use that motivated their construction, and as such it is the most obvious example of their usefulness.

7.1 The Application

The distributed application in this example models a database system in which a central server process partitions a request – such as "add record," "delete record," or "find record" – into several child processes that can run concurrently on a network of machines. This system is dynamic because these child processes are created at the time of each request and terminate independently when their respective work for this request is done. See appendices A, B, and C for the code for the example application.

The child processes all need to read and write shared data in the database. To prevent inconsistencies in the data due to concurrent access they need some form of synchronization to provide *critical sections* where only one process at a time can read or write to the data. This synchronization is provided by a *token-passing* protocol, where a single token circulates in a ring among the processes. Each process can enter its critical section if and only if it "has" the token – that is, if it has received an instance of a message containing the token but has not yet sent such a message to the next process in the ring. Figure 7.1 depicts the central server process and the child processes $C_1, C_2, C_3 \ldots C_n$ in the token-passing ring. The arrows between child processes denote the direction in which the token flows. In the pictured configuration, C_2 has just sent the token to C_3, which can now enter its critical section. When C_3 is done with its critical section it will send the token to C_4.

Intuitively, one can see that token-passing works because there is only one token and only the process with the token can enter its critical section. More specifically, token-passing ensures that two critical sections are never concurrent because there must be at least one message send and one message receive (namely, the token) between the two critical sections. Figure 7.2 shows how sending a token message naturally induces a happened-before relationship between the critical sections of processes. In this figure the critical section in P_1 between events e_1 and e_2 happens before the critical section in P_2 between events e_5 and e_6 because token is carried in a message that is sent in e_3 and received in e_4.

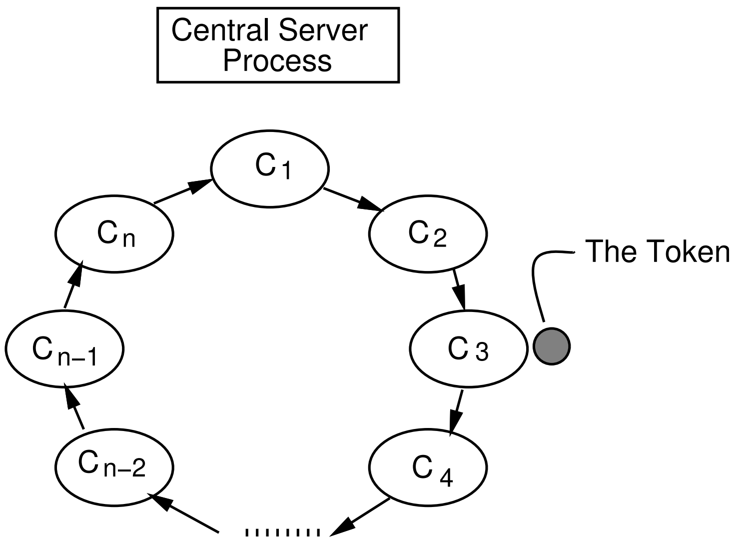

Figure 7.1: Token-Passing in the Example Application

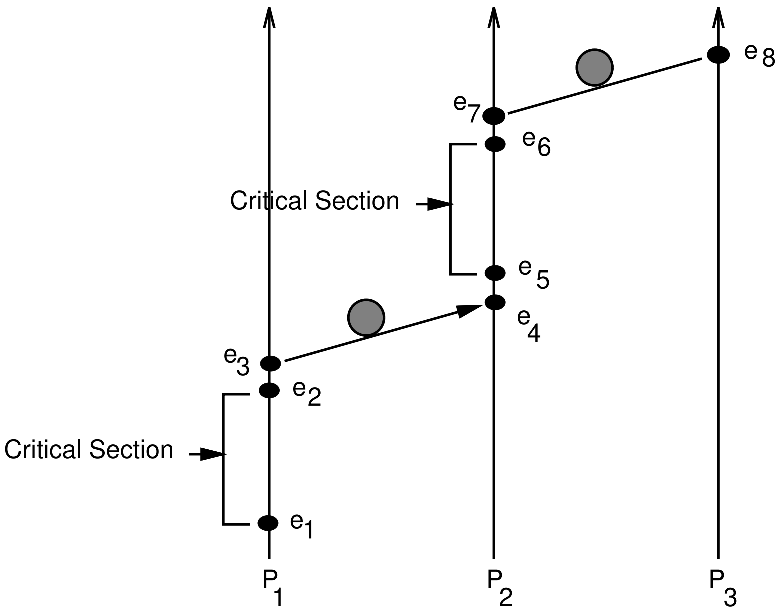

Figure 7.2: Token-Passing and Causality

7.2 The Bug

In order to test the usefulness of causally-structured output streams in debugging this distributed system, a typical error was introduced into the token-passing protocol.

In the correctly written child process, the code responsible for waiting for a token continued to be:

```
pvm_recv( -1, TOKEN );
```

This call results in the child process checking the in-bound message buffer for a message tagged as a `TOKEN` from any other task (thus the `-1`), and waiting for such a message if none is available.

In some child processes, this line was modified to the following nearly-identical line:

```
pvm_nrecv( -1, TOKEN );
```

This function is the *non-blocking* version of the above call. It makes the same check for a message tagged as a `TOKEN` but returns immediately if no such message has been received. This results in these processes thinking that they have received a token message immediately every time they intend to wait for a token. Thus whether or not they have *actually* received a token message they will continue into their critical sections.

This bug results in the erroneous children entering their critical section without regard to the progress of other processes. In a real application written to optimize performance at the sake of fault-tolerance this error could result in any manner of unexpected and incorrect behavior. The example application was designed so that it immediately discovers that some other process is in its critical section, notifies its predecessor and successor in the ring of its imminent departure, and terminates.

By printing an error message when this occurs it is relatively easy to ascertain that two processes have entered their critical sections simultaneously. The difficulty is in determining exactly which processes are at fault, and the sequence of events that lead to the error.

7.3 Using Causally-Structured Output for Debugging

In order to isolate the bug, the application was instrumented with causally-structured stream output. This was a simple undertaking. Even in large application this may be as simple as mechanically converting the use of regular output streams to causally-structured output streams.

The example application was modified so that the most important information – information about the entry and exit of critical sections – was written to one stream. Information about other aspects of the computation was directed to other streams: task spawning, task initialization, task termination, token sends, token receives, and reconfiguration of the token-passing ring upon task termination. Separating the important information into its own stream allowed the stream-viewing tool to eliminate distractions by displaying only the relevant stream. When more details are needed a programmer can ask SV to display the other output streams produced by the computation.

The programmer viewed the main output stream of an erroneous execution in SV(see figure 7.3). The top-left-most entry says that task `t40005` is entering its critical section. Below this entry is a message from task `t40005` to the effect that "some other process is already in its critical section." A simple click on the first entry clearly indicated the three concurrent entries at the top right by coloring them yellow. These three entries documented that task `t40003` entered its critical section, accessed the shared data, and left the critical section. The fact that `t40005` and `t40003` entered their critical section concurrently was clearly the source of the error.

The only remaining question was how both `t40005` and `t40003` thought that they had received a token. By expanding the view to include other streams, the programmer was able to examine the stream output entry from the central server process that announced that the server was about to send the token. This entry causally preceded the entries from `t40003`, but not the entries from `t40005`, indicating that `t40005` *hadn't actually received a message from the server.* At this point it was clear that there was a bug in the code for task `t40005` such that it thought it received a token without actually receiving a token. This suggested that the error was in the modified line in `t40005`:

```
pvm_nrecv( -1, TOKEN );
```

When the erroneous line is changed to its correct form, the application's token-passing protocol functions correctly, as depicted in figure 7.4. In this system each process waits for the token, enters its critical section, accesses some shared data, leaves its critical section, and passes the token to its

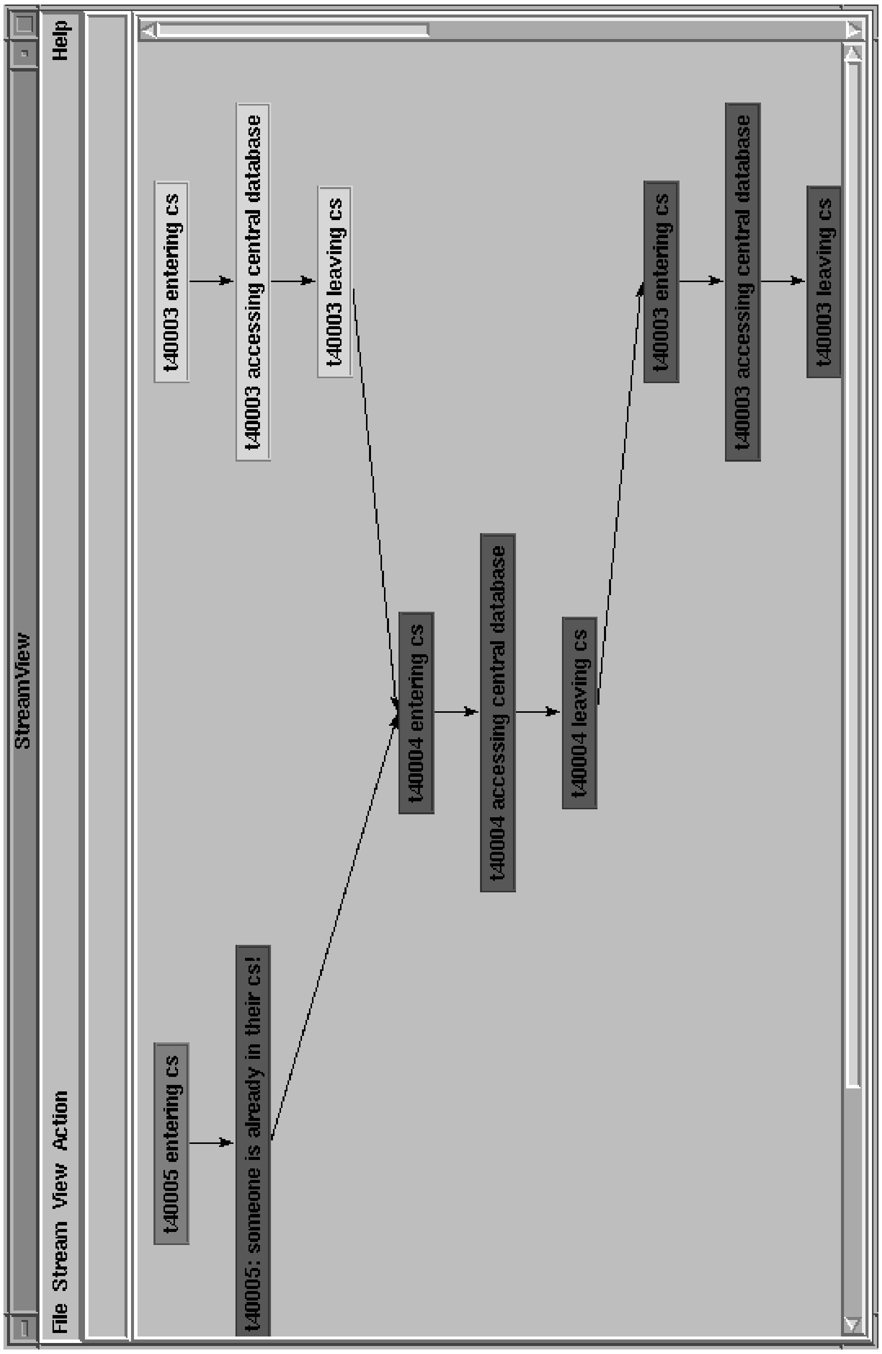

Figure 7.3: Token-Passing: Incorrect Functioning

neighbor. This results in an orderly progression of the token from task to task. As a result, the accesses to the database are neatly serialized. This figure resembles figure 7.2, as well it should, since they depict identical token-passing protocols.

7.4 Conclusion

This example shows how causally-structured output streams can provide a simple but powerful tool for isolating synchronization errors in distributed applications. By the simple process of instrumenting an application with causally-structured stream output a programmer can use SV to gain a dramatic visualization of the causal structure of a computation.

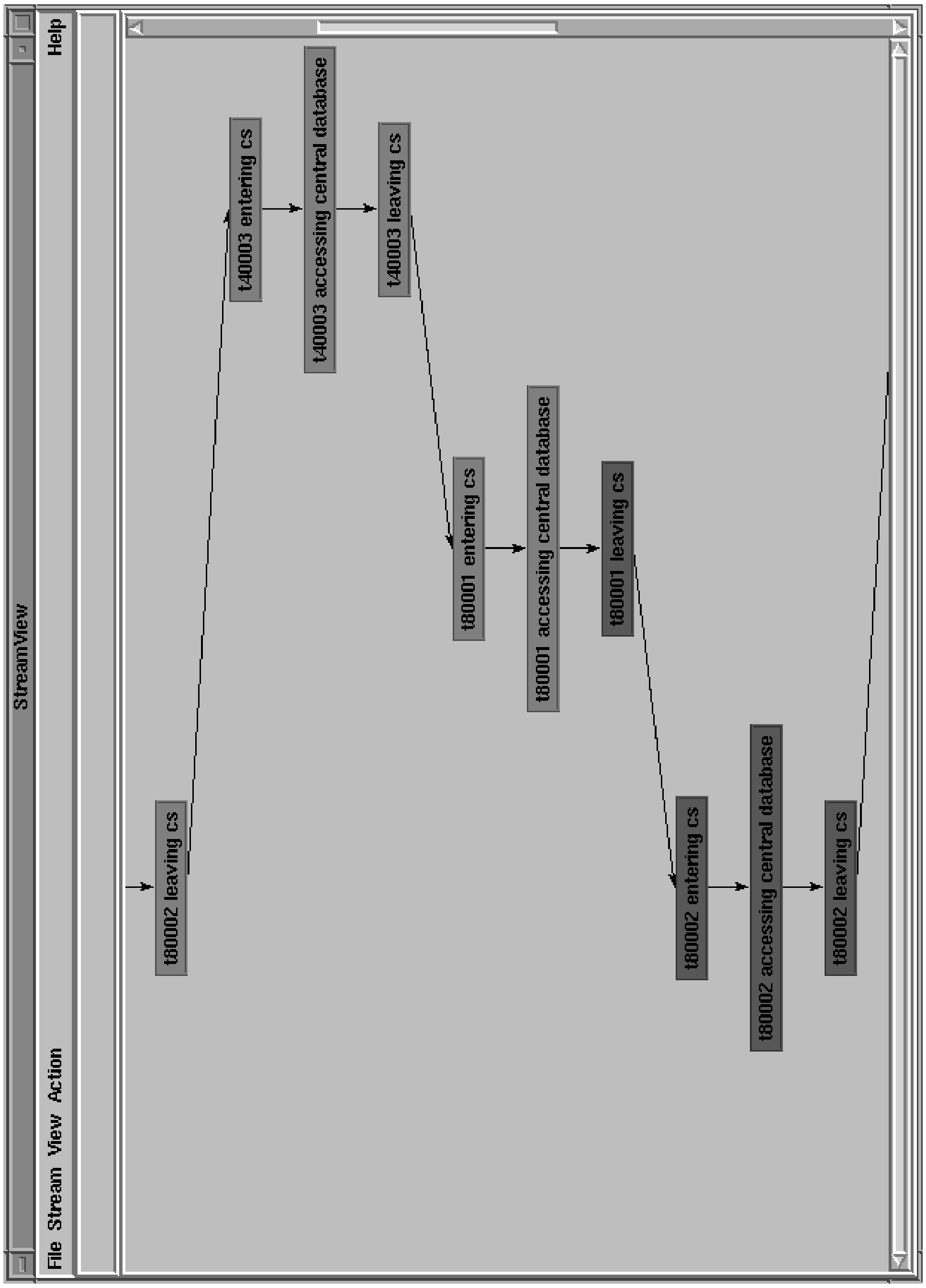

Figure 7.4: Token-Passing: Correct Functioning

Chapter 8

Conclusion

This paper defined the concept of causally-structured output streams for distributed systems as a straightforward extension of sequential output streams. The simple chronological ordering scheme of sequential output streams is replaced by Lamport's happened-before partial order.

It also described an implementation of causally-structured output streams for the PVM message-passing library which solves three of the main problems in distributed output streams: combining output from different hosts in the distributed system, maintaining the identity of entries in the output stream, and specifying a clear and useful order for entries in a distributed output stream.

It presented a graphical tool that can be used to produce useful visualizations of output streams and the causal relationships between entries in the output stream.

Finally it presented a case study showing how causally-structured output streams can be valuable in debugging distributed applications with synchronization errors.

Future work in this direction could attempt to reduce the overhead of time stamps in vector time implementation still further, specify and implement a more compact format for trace files, design a faster protocol for reading trace files, or discover algorithms for efficiently displaying large causally-structured streams.

Appendix A

Example Application – Code for Master Server

```
/* @(#)dbmaster.c 1.5 */

#include <stdio.h>
#include <assert.h>
#include "pvm3.h"

#include "dbproto.h"

#define SLAVENAME "dbslave"
#define OUT PVM_STDOUT


int
main(int argc, char *argv[])
{
    int mytid;                  /* my task id */
    int tids[32];               /* slave task ids */
    int n, nproc, numt, i, nhost, narch;
    float data[100], result[32];
    struct pvmhostinfo *hostp[32];
    int numtoadd;
    int cc;
```

```
/* enroll in pvm */
mytid = pvm_mytid();

cc = pvm_sopen(OUT);

/* Set number of slaves to start */
/* Can not do stdin from spawned task */
if( pvm_parent() == PvmNoParent ){
   printf("[t%x] How many slave programs (1-32)?",
          mytid);
   scanf("%d", &nproc);
}
else{
   pvm_config( &nhost, &narch, hostp );
   nproc = nhost;
}

if( nproc > 32 ) nproc = 32 ;

pvm_event();

cc = pvm_sprintf(OUT, "spawning %d slave tasks", nproc);

/* start up slave tasks */
numt = pvm_spawn(SLAVENAME,
                 (char**)0, 0, "",
                 nproc, tids);

if( numt < nproc ){
   printf("Trouble spawning slaves. Error codes:\n");
   for( i=numt ; i<nproc ; i++ ) {
      printf("TID %d %d\n",i,tids[i]);
   }
   for( i=0 ; i<numt ; i++ ){
      pvm_kill( tids[i] );
   }
   pvm_exit();
   exit();
}
```

```
/* tell the slaves what to do, who is their pred, succ */
for (i = 0; i < nproc; i++ ){
  int job, pred, succ;

  if (i == 0) {
    job  = ADD_RECORD;
    pred = tids[nproc - 1];
    succ = tids[i + 1];
  }
  else if (i == nproc - 1) {
    job  = DELETE_RECORD;  /* Last process is buggy!!! */
    pred = tids[i - 1];
    succ = tids[0];
  }
  else {
    job  = ADD_RECORD;
    pred = tids[i - 1];
    succ = tids[i + 1];
  }

  cc = pvm_sprintf(OUT,
                   "telling t%X pred=t%X, succ=t%X, job=%d",
                   tids[i], pred, succ, job);

  pvm_initsend( PvmDataDefault );
  pvm_pkint( &job,  1, 1 );
  pvm_pkint( &pred, 1, 1 );
  pvm_pkint( &succ, 1, 1 );
  pvm_send( tids[i], REQUEST );
}

/* pass a token to first task to get the ball rolling */
pvm_initsend( PvmDataDefault );
pvm_send( tids[0], TOKEN );

/* Wait for results from slaves */
for (i = 0; i < nproc; i++){
  int tid;

  /* will need to pass along the token? */
```

```
      pvm_recv( -1, TERM_MESG );
      pvm_upkint( &tid, 1, 1 );

      pvm_event();
      printf("child t%X finished\n", tid);

      cc = pvm_sprintf(OUT, "child t%X finished", tid );

    }

    cc = pvm_sprintf(OUT, "done!");
    cc = pvm_sclose(OUT);

    /* Program Finished -- exit PVM before stopping */
    pvm_exit();

    return 0;
}
```

Appendix B

Example Application – Code for Slave Children

```
/* @(#)dbslave.c 1.6 */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "pvm3.h"

#include "dbproto.h"

#define OUT 1
#define CSSTREAM 2

#define WORK_TO_DO   3

char lockname[200] = "/home/scratch2/ndcard/mutex";

int mytid;                /* my task id */
int pred;                 /* tid of predecessor in the ring */
int succ;                 /* tid of successor in the ring */
int job;
```

```
int workdone = 0;         /* number of cs-entries */

/* ------------------------------------------------------- */

void
get_token()
{
  /* wait for the token to come to us */

  if (job == DELETE_RECORD)
    pvm_nrecv( -1, TOKEN);  /* non-blocking!!! BUG!!! */
  else
    pvm_recv( -1, TOKEN);

  pvm_sprintf(OUT, "t%X has the token now", mytid);
}

void
send_token()
{
  check_config(); /* make sure we have our latest successor */
  pvm_sprintf(OUT, "t%X sending the token now", mytid);
  pvm_initsend( PvmDataDefault );
  pvm_send( succ, TOKEN);   /* send token */
}

void
pass_token()
{
  /* be a good neighbor and pass it along if we have it */
  int bufid;
  bufid = pvm_nrecv( -1, TOKEN );
  if (bufid > 0) {
    /* we've got a token, so pass it along */
    pvm_sprintf(OUT, "t%X got token but doesn't need it",
                mytid);
    send_token();
  }
}
```

```
int
check_config()
{
  /* see if succ has changed. it may have changed lots. */
  int bufid;
  bufid = pvm_nrecv( succ, NEWSUCC );
  while (bufid > 0) {
    /* our succ is leaving, so find out new successor */
    pvm_upkint(&succ, 1, 1);

    pvm_sprintf(OUT, "t%X has a new successor: t%X",
                mytid, succ);
    bufid = pvm_nrecv( succ, NEWSUCC );
  }

  bufid = pvm_nrecv( pred, NEWPRED );
  while (bufid > 0) {
    /* our predecessor is leaving, so find out new pred */
    pvm_upkint(&pred, 1, 1);

    pvm_sprintf(OUT, "t%X has a new predecessor: t%X",
                mytid, pred);
    bufid = pvm_nrecv( succ, NEWPRED );
  }

}

/* -------------------------------------------------------- */

int
enter_cs()
{
  get_token();

  pvm_sprintf(CSSTREAM, "t%X entering cs", mytid);
}

int
leave_cs()
{
```

```
  pvm_sprintf(CSSTREAM, "t%X leaving cs", mytid);

  send_token();
}

/* ------------------------------------------------------- */


void
startup()
{
  int cc;

  /* enroll in pvm */
  mytid = pvm_mytid();

  cc = pvm_sopen(OUT);
  cc = pvm_sopen(CSSTREAM);

  cc = pvm_sprintf(OUT,
                   "t%X waiting for msg from parent\n",
                   mytid);

  /* Receive assignment from master, place in ring */
  pvm_recv( pvm_parent(), REQUEST );
  pvm_upkint(&job, 1, 1);
  pvm_upkint(&pred, 1, 1);
  pvm_upkint(&succ, 1, 1);

  srand(mytid);

  cc = pvm_sprintf(OUT,
                   "t%X up: pred=t%X, succ=t%X, job=%d\n",
                   mytid, pred, succ, job);
}

void
teardown()
{
  int cc;
```

```
  cc = pvm_sprintf(OUT, "t%X waiting for token so can exit",
                   mytid);

  enter_cs();
  {
    cc = pvm_sprintf(OUT, "t%X informing pred t%X, succ t%X",
                     mytid, pred, succ);

    check_config();

    /* assumes FIFO between processes */

    /* inform predecessor of his new successor */
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &succ, 1, 1 );
    pvm_send( pred, NEWSUCC);

    /* inform successor of new predecessor */
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &pred, 1, 1 );
    pvm_send( succ, NEWPRED);

  }
  leave_cs();

  cc = pvm_sprintf(OUT, "t%X sending notification to parent",
                   mytid);

  /* Send result to master */
  pvm_initsend( PvmDataDefault );
  pvm_pkint( &mytid, 1, 1 );
  pvm_send( pvm_parent(), TERM_MESG );

  cc = pvm_sclose(OUT);
  cc = pvm_sclose(CSSTREAM);

  /* Program finished. Exit PVM before stopping */
  pvm_exit();
}
```

```
/* ------------------------------------------------------- */

int
want_cs()
{
  return (rand() % 100) < 25;
}


int
do_plain_stuff()
{
  /* stuff that doesn't require a critical section */
}

int
do_critical_stuff()
{
  struct stat info;
  int cc;
  int sleep(int seconds);

  /* check to make sure we're the only ones in here */
  if (stat(lockname, &info) >= 0)
    {
      pvm_sprintf(CSSTREAM,
                  "t%X: someone is already in their cs!",
                  mytid);

      cc = pvm_sprintf(OUT,
                       "t%X informing pred t%X, succ t%X",
                       mytid, pred, succ);

      check_config();

      /* assumes FIFO between processes */

      /* inform predecessor of his new successor */
      pvm_initsend( PvmDataDefault );
```

```
    pvm_pkint( &succ, 1, 1 );
    pvm_send( pred, NEWSUCC);

    /* inform successor of new predecessor */
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &pred, 1, 1 );
    pvm_send( succ, NEWPRED);

    /* Send result to master */
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &mytid, 1, 1 );
    pvm_send( pvm_parent(), TERM_MESG );

    cc = pvm_sclose(OUT);
    cc = pvm_sclose(CSSTREAM);

    pvm_exit();
    exit(-1);
  }

/* mark that we're in here */
cc = creat(lockname, 0);
assert(cc >= 0);

{

  workdone++;

  /* sleep to simulate computation */
  sleep( rand() % 3 + 1 );

  pvm_sprintf(CSSTREAM,
              "t%X accessing central database",
              mytid);

}

/* mark that we've left */
cc = unlink(lockname);
assert(cc >= 0);
```

```
}


void
loop()
{
  while ( workdone < WORK_TO_DO ) {
    if (want_cs()) {
      enter_cs();
      do_critical_stuff();
      leave_cs();
    }
    else {
      do_plain_stuff();
      /* be a good neighbor and pass it along if need be */
      pass_token();
    }
  }
}

/* -------------------------------------------------------- */

main()
{
  startup();
  loop();
  teardown();
}
```

Appendix C

Example Application – Interface for Protocols

```
/* @(#)dbproto.h 1.1 */

/* master-slave protocol */
#define REQUEST       0
#define ADD_RECORD    1
#define DELETE_RECORD 2

/* slave-master protocol */
#define TERM_MESG     5

/* slave-slave protocol */
#define TOKEN   6
#define NEWSUCC 7
#define NEWPRED 8
```

Bibliography

[1] *Proceedings of the ACM Symposium on High Level Debugging* (Aug. 1983), vol. 18 of *SIGPLAN Notices*.

[2] Edwards, D., and Kearns, P. DTVS: a distributed trace visualization system. In *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*. IEEE Comput. Soc. Press, Oct. 1994.

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

[4] Fidge, C. Fundamentals of distributed system observation. Tech. Rep. 93-15, University of Queensland, Nov. 1993.

[5] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., manchek, R., and Sunderan, V. *PVM: Parallel Virtual Machine*. The MIT Press, 1994.

[6] Joyce, J., Lomow, G., Slind, K., and Unger, B. Monitoring distributed systems. *ACM transactions on Computer Systems 5*, 2 (May 1987), 121–150.

[7] Lamport, L. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*. July 1978.

[8] LeBlanc, R. J., and Robbins, A. D. Event-driven monitoring of distributed programs. In *Proc. of the 5th International Conf. on Distributed Computing Systems*. IEEE, May 1985, pp. 515–522.

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

[10] Miller, B. P., and McDowell, C., Eds. *Summary of ACM/ONR Workshop on Parallel and Distributed Debugging* (1993), vol. 27 of *Operating Sysetsms Review*.

[11] Ousterhout, J. K. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[12] Paul K. Harter, J., Heimbigner, D. M., and King, R. IDD: An interactive distributed debugger. In *Proc. of the 5th International Conf. on Distributed Computing Systems*. IEEE, May 1985, pp. 498–506.

[13] Topol, B., Stasko, J., and Sunderam, V. Integrating visualization support into distributed computing systems. In *Proceedings of the 15th International Conference on Distributed Computing Systems*. June 1995.

[14] Walker, D., and Dongarra, J. MPI: A standard message passing interface. *Supercomputer 12*, 1 (Jan 1996), 56–68.