IEEE Region 10 Conference, Tencon 92
11th - 13th November, 1992
Melbourne, Australia

IMPROVEMENTS ON FIDGE LOGICAL CLOCKS

Wing-Hong Cheung
Department of Computer Science
University of Hong Kong
Pokfulam Road
Hong Kong
whcheung@csd.hku.hk

Abstract

Fidge introduced a logical clock mechanism for maintaining the partial ordering of events in distributed systems. However, the mechanism handles events in a rather ad hoc way and its comparison rule is not quite simple. In this paper, two major improvements on the mechanism are presented. Firstly, general maintenance rules are developed to handle various kinds of interprocess events. Secondly, the comparison rule is simplified by maintaining the clocks with a new technique called *two-phase update*. These improvements make the logical clock mechanism more general and efficient for supporting distributed system development.

1 Introduction

Event ordering is a very important tool for researchers to study, analysis and control the behaviour of a distributed program. In the paper [Lam78], Lamport defined the "happened-before" relation ($\rightarrow$) to determine the order of distributed events and introduced a logical clock mechanism to timestamp events according to the happened-before relation. However, his logical clock mechanism can be used to construct only an interleaved total ordering on the events. This basically gives a simple view of distributed computations and does not provide precise information of the happened-before relation between events.

Therefore, in the last decade, many people attempted to improve and extend Lamport clocks to solve problems in distributed system development. A common feature of these extensions is the use of time vectors, instead of an integer value, to represent logical time. For example, Gorden developed a time vector mechanism to keep trace of message passing events, and used the vector to detect misordered messages [Gor85, Section 2.2]. Choi *et al.* proposed an algorithm with two time vectors to determine the precedence ordering of events for analyzing concurrent events [CMN88, Section 6]. However, most of the proposed mechanisms are rather ad hoc and complicated. In 1989, Fidge introduced a more general logical clock mechanism to maintain the precise partial ordering of events [Fid88]. He further improved the mechanism to support distributed programs with dynamic process creation and termination and to unify the two comparison rules for asynchronous and synchronous communications into one [Fid91]. A brief review of logical clocks for distributed systems is given in [Ray92].

In this paper, two major improvements on Fidge clocks are presented. The first improvement is to generalize interprocess events into two generic classes in order to avoid specifying individual clock maintenance rules for each kind of events. This not only reduces the number of maintenance rules

0-7803-0849-2/92 $3.00 © 1992 IEEE

significantly, but also facilitates the support of a wide range of communication paradigms. The second improvement is to simplify the comparison rule of determining the happened-before relation. To achieve this, a new technique called *two-phase update* is used to timestamp events. This improvement is particularly helpful in manipulating a large amount of event timestamps, such as tracing events, computing program concurrency, studying and debugging concurrent behaviour. In Section 2, we highlight the two improvements on the Fidge logical clock mechanism. In Section 3, the detailed maintenance and comparison rules of the improved logical clocks are described. Finally, we conclude the discussion in Section 4.

2 The Two Improvements

Event Generalization. The development of the Fidge clock mechanism is based on the CSP model, and the interprocess events handled by it are restricted to only asynchronous and synchronous message passing and nested process creation and termination. However, for a truely general logical clock system, it should easily be incorporated or extended to different kinds of interprocess events and various interprocess communication paradigms.

The first improvement is to generalize all interprocess events into two major classes: *asynchronous* and *synchronous* interprocess events. This generalization is based on the following arguments:

- From many surveys of communication paradigms, asynchronous and asynchronous message passing primitives are the most common and basic ones. Other communication paradigms can be built on top of either one of them or both.
- Shared memory accesses with explicit synchronization tools (*e.g.*, semaphores) can be treated as synchronous events among processes and shared memory objects, as demonstrated in the development of Instant Replay [LMC87].
- Process events such as process creation and merged termination can be viewed as some kinds of "virtual" interprocess events, since they implicitly introduce causal relationship between the parent process and child process. In the case of process creation, the operation is as if a virtual asynchronous control message were sent from the creator to the created process to instruct the latter to execute. In most systems, a process termination event is a local event. However, for systems with process fork creation and merged termination operations (*e.g.*, CSP processes), the creator has to synchronize with the termination of its created processes. Hence, such a process termination event is basically an interprocess event.
- The support of multiple levels of communication abstraction is also an important reason to do event generalization. For examples, remote procedure call (RPC) can be modeled as a pair of asynchronous message interactions: one for procedure call and one for return status. Moreover, from the user's viewpoint, a remote message passing interaction just involves two events: send and receive. However, from the OS designer's viewpoint, the interaction actually involves several asynchronous packet exchanges between two remote kernels. Indeed, all high-level communication paradigms are built on basic network asynchronous or synchronous communications. The maintenance rules in the improved clock mechanism are general enough to facilitate its setup to support various levels of communication abstraction.

In summary, in the improved mechanism, all interprocess events are generalized into two classes: *asynchronous event sets* and *synchronous event sets.* For a pair of asynchronous events e and f, if e is a *trigger event* (*e.g.*, send event) and f is a *response event* (*e.g.*, receive event), then $e \rightarrow f$. For a pair of synchronous events e and f, if there is an event g such that $e \rightarrow g$, then $f \rightarrow g$. Also, if there is an event h such that $h \rightarrow e$, then $h \rightarrow f$. Note that each event set may involve more than two interprocess events. This allows the clock mechanism to support broadcast and multicast communications.

Comparison Rule Simplification. In his first paper [Fid88], Fidge presented two individual comparison rules for synchronous and asynchronous message passing events. In the most recent paper [Fid91], he unifies the two comparison rules into one. The unified rule requires to compare two pairs of time values to determine the happened-before relation. In more detail, assume that a partially-ordered clock reading for an event of a process p_i is represented by a set of pairs $\{(i,n),\ldots,(l,n)\}$. Here, the first element of each pair represents a process identifier j and the second element represents a numerical clock value of p_j as perceived by p_i. If a process identifier is not in the set, its default clock value is zero. Then, given two timestamps t_{e_j} and t_{f_j} for event e_i and f_j, Fidge gives the following comparison rule:

$$e_i \rightarrow f_j \Leftrightarrow (t_{e_i}(i) \leq t_{f_j}(i)) \wedge (t_{e_i}(j) < t_{f_j}(j))$$

The second improvement is to simplify the comparison rule so that the checking involves only one pair of time values. More precisely, the improved comparison rule is:

$$e_i \rightarrow f_j \Leftrightarrow t_{e_i}(i) < t_{f_j}(i)$$

To achieve this, the maintenance rules are slightly modified. A new technique called *two-phase update* is used to maintain the logical clocks for interprocess events. The basic idea is to divide the clock update operation into two phases. In the first phase, the clock is updated to timestamp the current interprocess event. After the timestamp operation, the clock is adjusted in the second phase to prepare for the next event. The extra overhead is just one addition operation for each involved process in the event set. In return, we eliminate one logical-and and one timestamp comparison operation. The resulting comparison rule is simpler for users to interpret and use, and it speeds up timestamp comparison operations.

3 The Maintenance Rules of The Improved Mechanism

To maintain a partially-ordered logical clocks in a distributed program, each process instance p_i must maintain a set variable c_i to keep the current partially-ordered time as perceived by itself. When the process begins execution, c_i is initialized to the empty set. The value of c_i is then updated in a monotonically increasing manner. The updates are governed by the following maintenance rules:

Rule A: *Local event ticking.* Whenever a process instance p_i performs an event, it increments $c_i(i)$ at least once. This ticking is performed before other rules are applied.

Rule B: *Synchronous event set.* During the occurrence of a synchronous event set, all process involved $p_j,\ldots,p_l$ exchange their current clock values, and then maximize their local clocks using the clock values from every other participating process instance; that is, $\forall x : \{j\ldots l\}$, $c_x := \max(c_j,\ldots,c_l)$. Timestamp is made at this point, if any. After this, each of the process instance p_x increments the local clock values of all involved process instances except itself; that is, $\forall y : \{j\ldots l, y \neq x\}\ c_x(y) := c_x(y) + 1$. (The two-phase update technique is applied.)

Rule C: *Asynchronous event set.* Whenever a process instance p_i triggers an asynchronous interprocess event, it takes the current clock value as the timestamp of the event. Then, it puts the value of c_i into a message as t_i, increments $t_i(i)$ by one, and delivers the message to the corresponding response process(es). (The idea of two-phase update is applied. However, the second update is for the time value delivered in the message rather than the local clock value.)

Upon receiving the time value t_i from process instance p_i, the response process instance p_j maximizes its counter values using t_i; that is, $c_j := \max(t_i, c_j)$.

Comparison Rule. Given two timestamps t_{e_i} and t_{f_j}, we can determine the happened-before relation using the following rule:

$$e_i \rightarrow f_j \Leftrightarrow t_{e_i}(i) < t_{f_j}(i)$$

By case analysis, we can formally prove that the comparison rule is correct. Informally, one can observe that at every point of interprocess interactions and only at these points, we maximize the involving timestamps and increment the appropriate clock elements to maintain the "<" relation. Moreover, any scenario in which we assume the timestamp comparison condition holds without any causal relationship will lead to a contradiction.

An example. The example presented in [Fid91] is used to illustrate the improved clock mechanism so that the readers can perceive more easily the differences between the original and the improved mechanisms. In Figure 1, dots represent the events and circles represent the point where the current clock value is given next to it. We treat process creation and process merged termination as asynchronous event sets. Events G and C are asynchronous send and receive events respectively. Events F and I are synchronous message passing event pair.

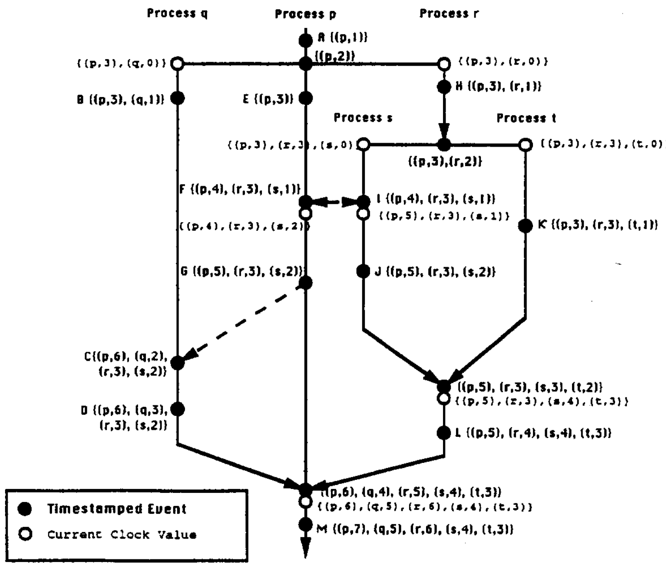

Figure 1: An Example of Timestamping Events with the Improved Logical Clock Mechanism

With the comparison rule, we have

- $\neg(C \to C)$ since $(2 \not< 2)$
- $B \to D$ since $(1 < 3)$
- $\neg(D \to B)$ since $(3 \not< 1)$
- $H \to G$ since $(1 < 3)$
- $\neg(C \to H)$ since $(2 \not< 0)$
- D and J are potentially concurrent because $\neg(D \to J)$ and $\neg(J \to D)$ (since $(3 \not< 0) \wedge (2 \not< 2)$).

4 Conclusion

In the past, people have typically studied distributed computations with an interleaved total ordering on the events performed. With the introduction of Fidge logical clocks, people have a basic tool to study distributed computations with the precise partial ordering of its events. In this paper, an improved version of this basic tool is presented. The improved mechanism provides generic and less number of maintenance rules to support various kinds of interprocess events and to simplify the comparison rule. These improvements facilitate the use of the clock mechanism in different communication paradigms and allows efficient timestamp comparisons. The improved mechanism has been implemented in a visualization tool to study interprocess events of concurrent programs [FL92]. The implementation is found to be effective and efficient.

References

[CMN88] J.D. Choi, B.P. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *Technical Report 786, Computer Sciences Department, University of Wisconsin-Madison*, August 1988.

[Fid88] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, February 1988.

[Fid91] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

[FL92] C.Y. Fung and K.Y. Leung. To design and implement a concurrent program execution visualization tool. *Final Year Project Report, Department of Computer Science, The University of Hong Kong*, March 1992.

[Gor85] A.J. Gordon. Ordering errors in distributed programs. *Ph.D Thesis, Technical Report 611, Computer Science Department, University of Wisconsin-Madison*, August 1985.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LMC87] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[Ray92] M. Raynal. About logical clocks for distributed systems. *ACM Operating System Review*, 26(1):41–48, January 1992.