Logical Time: A Way to Capture Causality in Distributed Systems

Michel Raynal, Mukesh Singhal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Logical Time: A Way to Capture Causality in Distributed Systems

M. Raynal, M. Singhal

N° 2472

Mars 1995

PROGRAMME 1

ISSN 0249-6399

Logical Time: A Way to Capture Causality in Distributed Systems

M. Raynal*, M. Singhal**

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Rapport de recherche n°2472 — Mars 1995 — 22 pages

Abstract: The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems. Usually causality is tracked using physical time, but in distributed systems setting, there is no built-in physical time and it is only possible to realize an approximation of it. As asynchronous distributed computations make progress in spurts, it turns out that the logical time, which advances in jumps, is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems. This paper reviews three ways to define logical time (e.g., scalar time, vector time, and matrix time) that have been proposed to capture causality between events of a distributed computation.

Key-words: Distributed systems, causality, logical time, happens before, scalar time, vector time, matrix time.

(Résumé : tsvp)

*IRISA, Campus de Beaulieu, 35042 Rennes-Cédex, raynal@irisa.fr.
**Dept. of Computer, Information Science, Columbus, OH 43210, singhal@cis.ohio-state.edu.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Le temps logique en réparti ou comment capturer la causalité

Résumé : Ce rapport examine différents mécanismes d'horlogerie logique qui ont été proposés pour capturer la relation de causalité entre événements d'un système réparti.

Mots-clé : causalité, précédence, sytème réparti, temps logique, temps linéaire, temps vectoriel, temps matriciel.

1 Introduction

A distributed computation consists of a set of processes that cooperate and compete to achieve a common goal. These processes do not share a common global memory and communicate solely by passing messages over a communication network. The communication delay is finite but unpredictable. The actions of a process are modeled as three types of events, namely, internal events, message send events, and message receive events. An internal event only affects the process at which it occurs, and the events at a process are linearly ordered by their order of occurrence. Moreover, send and receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process. It follows that the execution of a distributed application results in a set of distributed events produced by the processes. The causal precedence relation induces a partial order on the events of a distributed computation.

Causality (or the causal precedence relation) among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation. The knowledge of the causal precedence relation among processes helps solve a variety of problems in distributed systems. Among them we find:

- Distributed algorithms design: The knowledge of the causal precedence relation among events helps ensure liveness and fairness in mutual exclusion algorithms, helps maintain consistency in replicated databases, and helps design correct deadlock detection algorithms to avoid phantom and undetected deadlocks.
- Tracking of dependent events: In distributed debugging, the knowledge of the causal dependency among events helps construct a consistent state for resuming reexecution; in failure recovery, it helps build a checkpoint; in replicated databases, it aids in the detection of file inconsistencies in case of a network partitioning.
- Knowledge about the progress: The knowledge of the causal dependency among events helps a process measure the progress of other processes in the distributed computation. This is useful in discarding obsolete information, garbage collection, and termination detection.
- Concurrency measure: The knowledge of how many events are causally dependent is useful in measuring the amount of concurrency in a computation. All events that are not causally related can be executed concurrently. Thus, an analysis of the causality in a computation gives an idea of the concurrency in the program.

The concept of causality is widely used by human beings, often unconsciously, in planning, scheduling, and execution of a chore or an enterprise, in determining infeasibility of a plan or the innocence of an accused. In day-today life, the global time to deduce causality relation is obtained from loosely synchronized clocks (i.e., wrist watches, wall clocks). However, in distributed computing systems, the rate of occurrence of events is several magnitudes higher and the event execution time is several magnitudes smaller; consequently, if the physical clocks are not precisely synchronized, the causality relation between events may not be accurately captured. However, in a distributed computation, progress is made in spurts and the interaction between processes occurs in spurts; consequently, it turns out that in a distributed computation, the causality relation between events produced by a program execution, and its fundamental monotonicity property, can be accurately captured by logical clocks.

In a system of logical clocks, every process has a logical clock that is advanced using a set of rules. Every event is assigned a timestamp and the causality relation between events can be generally inferred from their timestamps. The timestamps assigned to events obey the fundamental monotonicity property; that is, if an event a causally affects an event b, then the timestamp of a is smaller than the timestamp of b.

This paper first presents a general framework of a system of logical clocks in distributed systems and then discusses three ways to implement logical time in a distributed system. In the first method, the Lamport's scalar clocks, the time is represented by non-negative integers; in the second method, the time is represented by a vector of non-negative integers; in the third method, the time is represented as a matrix of non-negative integers. The rest of the paper is organized as follows: The next section presents a model of the execution of a distributed computation. Section 3 presents a general framework of logical clocks as a way to capture causality in a distributed computation. Sections 4 through 6 discuss three popular systems of logical clocks, namely, scalar, vector, and matrix clocks. Section 7 discusses efficient implementations of the systems of logical clocks. Finally Section 8 concludes the paper.

2 A Model of Distributed Executions

2.1 General Context

A distributed program is composed of a set of n asynchronous processes p_1, p_2, ..., p_i, ..., p_n that communicate by message passing over a communication network. The

processes do not share a global memory and communicate solely by passing messages. The communication delay is finite and unpredictable. Also, these processes do not share a global clock that is instantaneously accessible to these processes. Process execution and a message transfer are asynchronous – a process may execute an event spontaneously and a process sending a message does not wait for the delivery of the message to be complete.

2.2 Distributed Executions

The execution of process p_i produces a sequence of events e_i^0, e_i^1, ..., e_i^x, e_i^{x+1}, ... and is denoted by $\mathcal{H}_i$ where

$$\mathcal{H}_i = (h_i, \rightarrow_i)$$

h_i is the set of events produced by p_i and binary relation $\rightarrow_i$ defines a total order on these events. Relation $\rightarrow_i$ expresses causal dependencies among the events of p_i.

A relation $\rightarrow_{msg}$ is defined as follows. For every message m that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} receive(m).$$

Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of corresponding send and receive events.

A distributed execution of a set of processes is a partial order $\mathcal{H}$=(H, $\rightarrow$), where

$$H = \cup_i h_i \text{ and } \rightarrow = (\cup_i \rightarrow_i \cup \rightarrow_{msg})^+.$$

Relation $\rightarrow$ expresses causal dependencies among the events in the distributed execution of a set of processes. If $e_1 \rightarrow e_2$, then event e_2 is directly or transitively dependent on event e_1. If $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$, then events e_1 and e_2 are said to be concurrent and are denoted as $e_1 \parallel e_2$. Clearly, for any two events e_1 and e_2 in a distributed execution, $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$, or $e_1 \parallel e_2$.

Figure 1 shows the time diagram of a distributed execution involving three processes. A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer. In this execution, $a \rightarrow b$, $b \rightarrow d$, and $b \parallel c$.

2.3 Distributed Executions at an Observation Level

Generally, at a level or for an application, only few events are relevant. For example, in a checkpointing protocol, only local checkpoint events are relevant. Let R denote