Time, Clocks, and the Ordering of Events in a Distributed System

Lamport

July 1978

Summary

The paper shows a way of totally ordering events in distributed systems.

Distributed system

- A **process** is a sequence of totally ordered events, i.e., for any event a and b in a process, either a comes before b or b comes before a.
- A **distributed system** is a set of distinct and "spatially separated" processes that communicate by sending messages to each other.
- A system is distributed if the time it takes to send a message from one process to another is significant compared to the time interval between events in a single process. *Why is this distinction important?*

Review of ordering relations

Taken from "Introduction to Discrete Mathematics" by Hirschfelder and Hirschfelder.

- A binary relation $\diamond$ on a set A is **reflexive** if $a \diamond a$ for every $a \in A$.
- A binary relation $\diamond$ on a set A is **symmetric** if, whenever $a \diamond b$, then $b \diamond a$.
- A binary relation $\diamond$ on a set A is **transitive** if, whenever $x \diamond y$ and $y \diamond z$, then $x \diamond z$.
- A relation $\diamond$ on a set A is **antisymmetric** if $a \diamond b$ and $b \diamond a$ imply $a = b$.
- A **partial order relation** on a set A is a relation that is reflexive, antisymmetric, and transitive. The term **partial** is used to indicate that it is not necessary for any two elements of the set to be related in some way.
- A **total order** relation $\diamond$ on a set A is a partial order relation with the following additional property: If a and b are elements of A, then either $a \diamond b$ or $b \diamond a$, i.e., any two elements of the set are related in some way.
- Examples:
  - Relations $\leq$ and $\geq$ are partial order relations that are also total order relations on the set of integers.
  - Relation $<$ is not a partial order relation on the set of integers because it is not reflexive.

"Happened before" relation

It is assumed that sending or receiving a message is an event.

Definition. The relation $\rightarrow$ on the set of events of a system is the smallest relation satisfying the following three conditions:

1. If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two distinct events a and b are **concurrent** if $a \not\rightarrow b$ and $b \not\rightarrow a$. Assume $a \not\rightarrow a$ for any event a. So $\rightarrow$ is an **irreflexive partial ordering** on the set of all events in the system. The ordering is only partial because events can be concurrent in which case it is not known which event happened first.

One can also say that $a \rightarrow b$ means that a caused b to happen, e.g., the sending of a message causes the receipt of the same message. Events a and b are concurrent if they in no way can causally affect each other, e.g., events p_3 and q_3 in Fig. 1. So concurrent events do not necessarily have to occur at the same time. As long as there is no causality between them, they are concurrent.

Logical clocks

- No physical time is associated with the clocks. Clocks just assign numbers to events as they happen. The numbers assigned are the times at which events occurred.
- Clock $C_i \equiv$ a function which assigns a number $C_i\langle a\rangle$ to any event a in process P_i
- Clock $C \equiv$ a function which assigns a number $C\langle b\rangle$ to any event b in the system where $C\langle b\rangle = C_j\langle b\rangle$ if b is an event in process P_j. So C represents all the clocks in the system.
- For system events to be ordered correctly, the **Clock Condition** must be satisfied:

  $$\text{For any events } a, b\text{: if } a \rightarrow b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

  If event a happened before event b, then the clock value assigned to a is less than that assigned to b.
- The Clock Condition is satisfied if the following two conditions hold:

  C1. If a and b are events in process P_i and a comes before b, then $C_i\langle a\rangle < C_i\langle b\rangle$.

  C2. If a is the sending of a message by process P_i and b if the receipt of that message by process P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$.

  C1 means that there must be at least a clock tick between any two events in a process. C2 means that there must be at least a clock tick between the sending of a message by a process and its corresponding receipt by another process.
- System clocks must satisfy conditions C1 and C2 to satisfy the Clock Condition. For this to happen, system processes must obey the following implementation rules:

  IR1. Each process P_i increments C_i between any two successive events.

  IR2. a. If event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i\langle a\rangle$.

  b. Upon receiving message m, process P_j sets C_j greater than or equal to its current value and greater than T_m.

  IR1 causes events in a process to happen at different "logical" times and satisfies condition C1. IR2 causes process clocks to be synchronized and satisfies condition C2.

Total ordering of events

- A system of clocks that satisfy the Clock Condition can be used to totally order system events.
- To totally order the events in a system, the events are ordered according to their times of occurrence. In case two or more events occur at the same time, an arbitrary total ordering $\prec$ of processes is used. To do this, the relation $\Rightarrow$ is defined as follows:

  If a is an event in process P_i and b is an event in process P_j, then $a \Rightarrow b$ if and only if either:

  i. $C_i\langle a\rangle < C_j\langle b\rangle$ or
  ii. $C_i\langle a\rangle = C_j\langle b\rangle$ and $P_i \prec P_j$

  There is total ordering because for any two events in the system, it is clear which happened first.
- The total ordering of events is very useful for distributed system implementation.

The mutual exclusion problem

A system has a fixed number of processes which share one resource. The shared resource can be used by only one process at a time. Assume that the resource is initially allocated to one process. Find an algorithm that allocates the resource to a process and satisfies the following conditions:

I. A process using the resource must release it before it can be given to another process.

II. Requests for the resource must be granted in the order in which they were made. (This condition does not indicate what should be done when two processes request for the resource at the same time. Which process should get it?)

III. If every process using the resource *eventually* releases it, then every request is *eventually* granted (no starvation).

A distributed algorithm to solve the mutual exclusion problem

Assumptions

- For any two processes P_i and P_j, the messages sent from P_i to P_j are received in the same order they were sent.
- Every message is eventually received.
- A process can send messages directly to every other process in the system.

Algorithm

Let P_0 be the process to which the shared resource is initially allocated. Let T_0 be less than the initial value of any logical clock in the system. Each process has its own private request queue. Initially, each request queue contains one message, $T_0 : P_0$ *requests resource*. The following rules define the algorithm. Each rule is an event. Note that implementation rules IR1 and IR2 are used to maintain the process clocks.

1. *Resource request.* Process P_i sends the message $T_m : P_i$ *requests resource* to every other process where T_m is the process clock's value at the time of the request. P_i also puts the request message on its request queue.

2. *Resource request receipt.* P_j receives P_i's request message. P_j then puts the message on its request queue and sends an acknowledgement to P_i. By IR2, the the acknowledgement is timestamped later than T_m. If P_j has already sent P_i a message timestamped later than T_m, it doesn't have to send an acknowledgement since all that P_i needs is a message from P_j timestamped later than T_m.

3. *Resource release.* P_i removes request message $T_m : P_i$ *requests resource* from its queue and sends the release message P_i *releases resource* to every other process.

4. *Resource release receipt.* P_j receive's P_i's resource release message. P_j removes the $T_m : P_i$ *requests resource* from its request queue.

5. *Resource allocation.* P_i is allocated the resource when:

   (i) There is a $T_m : P_i$ *requests resource* message in P_i's request queue which is ordered before any other request in the queue by $\Rightarrow$.

   (ii) P_i has received messages from every other process timestamped later than T_m.

See paper on how the above rules satisfy conditions I—III of the mutual exclusion problem.

Anomalous behavior

The algorithm described above orders requests using the relation $\Rightarrow$. This can cause the following anomalous behavior to occur. Assume a system of interconnected computers across the country with some shared resource. A user issues request a on computer A to request for the shared resource. He then calls a friend in another city to issue request b on computer B to also request for that resource. Request b can be ordered before a on computer B if the request message for a is received by B after request b has been made. This causes computers A and B to have requests ordered differently. a comes before b on computer A and b comes before a on computer B. It can happen that at some point, a is the first request on computer A and b is the first on B. This satisfies condition (i) of the algorithm's rule 5 (resource allocation rule). Assume that condition (ii) has already been satisfied. Then both computers will try to obtain the shared resource at about the same time causing a conflict.

There are two possible ways to avoid such anomalous behavior:

1. Give the user the responsibility for avoiding anomalous behavior. For example, when request a is made, the user making the request is given the timestamp for that request. When that user then calls his friend, the friend can request that request b be given a later timestamp than a.

2. Let $\mathcal{S}$ be the set of all system events and $\underline{\mathcal{S}}$ be the set containing $\mathcal{S}$ along with relevant events external to the system. Let $\hookrightarrow$ denote the "happened before" relation for $\underline{\mathcal{S}}$. Construct a system of independent physical clocks that satisfies the following **Strong Clock Condition**:

$$\text{For any events } a, b \text{ in } \mathcal{S}\text{: if } a \hookrightarrow b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

Physical clocks

Let $C_i(t)$ denote the reading of clock C_i at physical time t. Assume that $C_i(t)$ is a continuous, differentiable function of t except for isolated discontinuities introduced by clock resets. Then $\frac{dC_i(t)}{dt}$ is the rate at which clock C_i is running at time t. To satisfy the Strong Clock Condition, the system of physical clocks must satisfy the following conditions:

PC1. There exists a constant $\kappa \ll 1$ such that for all i: $|\frac{dC_i(t)}{dt} - 1| < \kappa$.

PC2. For all $i, j : |C_i(t) - C_j(t)| < \epsilon$.

Condition PC1 says that the rate at which each physical clock C_i runs should vary only by a very small amount (bounded by κ). The paper assumes that PC1 is satisfied, i.e., that clocks run at approximately the correct rate. Condition PC2 says that for all the clocks in the system to be synchronized, their readings at time t should vary within a threshold amount (ϵ). See the paper for details on how to determine the values of κ and ϵ. To guarantee that PC2 is satisfied by the system of physical clocks, processes must obey the following implementation rules (specialization of IR1 and IR2). Let m be a message sent at physical time t and received at time $t\prime$. Let $v_m = t\prime - t$ be the *total delay* of m which is unknown to the receiving process. Let μ_m be some minimum delay ≥ 0 known by the receiving process such that $\mu_m \leq v_m$. Let $\xi_m = v_m - \mu_m$ be the *unpredictable delay* of m. Assume that each event occurs at a specific instant of physical time and that events of a process occur at different times.

IR1$\prime$. For each i, if P_i does not receive a message at physical time t, then C_i is differentiable at t and $\frac{dC(t)}{dt} > 0$.

IR2$\prime$. a. If P_i sends a message m at physical time t, then m contains a timestamp $T_m = C_i(t)$.

b. Upon receiving a message m at time $t\prime$, process P_j sets $C_j(t\prime)$ equal to $\max(C_j(t\prime - 0), T_m + \mu_m)$.

$IR1\prime$ states that clock readings change with physical time. $IR2\prime$ states how the clocks synchronize with each other (and that this synchronization is coupled with message receipts since this is the only way that processes can communciate with each other). P_j's clock is set to either its current time or the time at which the message is sent plus the expected minimum transmission delay, whichever is greater.

See paper for the theorem (and its proof) which states that $IR1\prime$ and $IR2\prime$ establish PC2. The theorem also bounds the time it takes for the clocks to get synchronized at system startup time.