Distributed Software Development
Synchronization

Chris Brooks

Department of Computer Science
University of San Francisco
In systems with multiple CPUs, the clocks are unlikely to have the exact same time.

- Variations in manufacturing cause clock skew.

Insight: often, it doesn’t matter exactly what time an operation happens, but what order events occur in.

(exception: hard real-time systems)
4-1: Logical clocks

- A logical clock is just a counter (or set of counters)
- What’s important is that all processes in the system can use it to produce a consistent ordering.
- This lets all processes in the system construct a global view of the system state.
Many interesting problems in distributed computing can be expressed in terms of determining whether some property \( p \) is satisfied.

- Consensus
- Deadlock
- Termination
- Load balancing

The general version of this problem is called the *Global Predicate Evaluation Problem*.
For example, to detect deadlock, we construct a wait-for graph.

- Edge from process $p_i$ to $p_j$ if $p_i$ is waiting on a message from $p_j$.

If this graph has a cycle, we have deadlock.

How do we do this while the computation is running?

We can’t pause the computation and collect all the information in one place.
We define a distributed computation to consist of a set of processes $p_1, p_2, \ldots, p_n$.

Unidirectional channels exist between each process for passing messages.

We assume these channels are reliable, but not necessarily FIFO.

- Messages may arrive out of order.

Assume the communication channels are asynchronous

- No shared clock
- No bound on message delay
Most of the time, we don’t necessarily care about the exact time when each event happens. Instead, we care about the order in which events happen on distributed machines. If we do care about time, then the problem becomes one of synchronizing about the global value of a clock.
A process $p_i$ consists of a series of events $e^1_i, e^2_i, \ldots$

There are three types of events:
- Local events - no communication with other processes
- Send events
- Receive events

A local history is a sequence of events $e^1_i, e^2_i, \ldots$ such that order is preserved.
The *initial prefix* of a history containing the first \( k \) events is denoted: 
\[ h^k_i = e^1_i, e^2_i, ..., e^k_i \]

\[ h^0_i = <> \]

The *Global history* of a computation is the union of all local histories.
\[ h_1 \cup h_2 \cup ... \cup h_n \]

Notice that this doesn’t say anything about order of events between processes.

Since an asynchronous system implies that there is no global time frame between events, we need some other way to order events on different processes.
Cause and effect can be used to produce a partial ordering.

Local events are ordered by identifier.

Send and receive events are ordered.
- If $p_1$ sends a message $m_1$ to $p_2$, \textit{send}(m_1) \text{ must occur before } \textit{receive}(m_1)$.
- Assume that messages are uniquely identified.

If two events do not influence each other, even indirectly, we won’t worry about their order.
The *happens before* relation is denoted →.

Happens before is defined:

- If \( e_i^k, e_i^l \) and \( k < l \), then \( e_i^k \rightarrow e_i^l \)

- (sequentially ordered events in the same process)

- If \( e_i = \text{send}(m) \) and \( e_j = \text{receive}(m) \), then \( e_i \rightarrow e_j \)

- (send must come before receive)

- If \( e \rightarrow e' \) and \( e' \rightarrow e'' \), then \( e \rightarrow e'' \)

- (transitivity)

If \( e \not\rightarrow e' \) and \( e' \not\rightarrow e \), then we say that \( e \) and \( e' \) are concurrent. \( (e \parallel e') \)

These events are unrelated, and could occur in either order.
Happens before provides a partial ordering over the global history. \((H, \rightarrow)\)

We call this a distributed computation.

A distributed computation can be represented with a space-time diagram.
4-11: Space-time diagram
6 Arrows indicate messages sent between processes.
6 Causal relation between events is easy to detect
6 Is there a directed path between events?
6 \( e_1 \rightarrow e_3^4 \)
6 \( e_1^2 \parallel e_3^1 \)
Let $\sigma_i^t$ represent the local state of process $p_i$ after executing event $e_i^t$.

$\sigma_i^0$ is the initial state of a process before executing an event.

The *global state* of a system is an $n$-tuple of local states, one for each process.

$\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$

A *cut* is a subset of the global history containing an initial prefix for each process.

Visually, the cut is a (possibly curved) line through the spacetime diagram.
The frontier of the cut is the last state in each process.

We’ll use a cut to try to specify the global state of a computation at some point in time ...

A run is a total ordering over events that is consistent with each local history.

△ A distributed computation can have many runs.
So how can we use all of this to solve the GPE problem?

We want to know what the global state of the system is at some point in time ...

Solution 1: Create $p_0$, the monitor process.

The monitor sends each process an 'inquiry' message.

Each process responds by sending its current local state $\sigma_i$

Once all local states are received, these define the frontier of a cut.

This cut is the global state. Will this work?
Recall that the monitor is a part of the computation, and subject to the same communication rules.

Let’s say we want to use this to do deadlock detection.

Each process can send a request and receive a reply.

If there is a cycle in the resulting wait-for graph (WFG), we have a deadlock.
4-17: Monitoring a distributed computation
4-18: Monitoring a distributed computation

Suppose \( p_1 \) receives the monitor message after \( e_1^3 \), \( p_2 \) after \( e_2^2 \), and \( p_3 \) after \( e_3^4 \).

The WFG has edges (1,3), (2,1), (3,2).

The system is not really deadlocked, though; the monitor received an inconsistent picture.

This is called a ghost deadlock.

Problem: process \( p_3 \)’s state reflects receiving a message that (according to the monitor) \( p_1 \) never sent.

Active monitoring isn’t going to work.
We need to restrict our monitor to looking at consistent cuts.

A cut is consistent if, for all events \( e \) and \( e' \)
\[ (e \in C \text{ and } e' \rightarrow e) \Rightarrow e' \in C \]

In other words, we retain causal ordering and preserve the 'happens before' relation.

A consistent cut produces a consistent global state.

A consistent run is one such that, for all \( e, e' \), if \( e \rightarrow e' \), then \( e \) appears before \( e' \) in the run.

This produces a series of consistent global states.

We need an algorithm that produces consistent cuts.
Let’s alter our monitor $p_0$.

- Rather than sending ’inquiry’ messages, it listens.
- Whenever any other process executes an event, it sends a message to $p_0$.
- $p_0$ would like to reconstruct a consistent run from these messages.

How to prevent out-of-order messages?
4-21: Synchronous communication

How could we solve this problem with synchronous communication and a global clock?

Assume FIFO delivery, delays are bounded by $\delta$

- $send(i) \rightarrow send(j) \Rightarrow deliver(i) \rightarrow deliver(j)$
- Receiver must buffer out-of-order messages.

Each event $e$ is stamped with the global clock: $RC(e)$.

When a process notifies $p_0$ of event $e$, it includes $RC(e)$ as a timestamp.

At time $t$, $p_0$ can process all messages with timestamps up to $t - \delta$ in increasing order.

No earlier message can arrive after this point.
4-22: Why does this work?

6 If we assume a delay of $\delta$, at time $t$, all messages sent before $t - \delta$ have arrived.

6 By processing them in increasing order, causality is preserved.

6 $e \rightarrow e' \Rightarrow RC'(e) < RC'(e')$

6 But we don't have a global clock!!
Each process maintains a logical clock. ($LC$).
Maps events to natural numbers. (0,1,2,3,...).
In the initial state, all LCs are 0.
Each message $m$ contains a timestamp indicating the logical clock of the sending process.

After each event, the logical clock for a process is updated as follows:
- $LC(e) = LC + 1$ if $e$ is a local or send event.
- $LC(e) = \max(LC, TS(m)) + 1$ if $e = receive(m)$.

The LC is updated to be greater than both the previous clock and the timestamp.
4-24: Logical clock example
Notice that logical clock values are increasing with respect to causal precedence.

- Whenever $e \rightarrow e'$, $LC(e) < LC(e')$

The monitor can process messages according to their logical clocks to always have a consistent global state.

Are we done?

- Not quite: this delivery rule lacks *liveness*.
- Without a bound on message delay, we can never be sure that we won’t have another message with a lower logical clock.
- We can’t detect gaps in the clock sequence.
- Example: we’ll wait forever for the nonexistent message for $e_3$ from $p_1$. 
We can get liveness by:

△ Delivering messages in FIFO order, buffering out-of-order messages.
△ Deliver messages in increasing timestamp order.
△ Deliver a message \( m \) from process \( p_i \) after at least one message from all other processes having a greater timestamp has been buffered.
Recall that FIFO only refers to messages sent by the same process.

causal delivery is a more general property which says that if \( \text{send}(m_1) \rightarrow \text{send}(m_2) \), then \( \text{deliver}(m_1) \rightarrow (m_2) \) when different processes are sending \( m_1 \) and \( m_2 \).

Logical clocks aren’t enough to give us causal delivery.
Solution: keep a “logical clock” for each process.

these are stored in a vector $VC$.

Assumes number of processes in known and fixed.

Update rule:

$VC'(e)[i] = VC[i] + 1$ for send and internal.

$VC'(e) = max(VC, TS(m))$ for receive; then

$VC'(e)[i] = VC[i] + 1$

On receive, the vector clock takes the max on a component-by-component basis, then updates the local clock.
**4-29: Vector Clock example**

Vector Clock example:

- $p_1$: $(1,0,0)$, $(2,1,0)$, $(3,1,3)$, $(4,1,3)$, $(5,1,3)$, $(6,1,3)$
- $p_2$: $(0,1,0)$, $(1,2,4)$, $(4,3,4)$
- $p_3$: $(0,0,1)$, $(1,0,2)$, $(1,0,3)$, $(1,0,4)$, $(1,0,5)$, $(5,1,6)$
The monitor can then process events in order of ascending vector clocks. This ensures causality.

Two clocks are inconsistent if $c_1[i] < c_2[i]$ and $c_1[i] > c_2[j]$.

If a cut contains no inconsistent clocks, it is consistent.

Vector clocks allow the implementation of causal delivery.
Cuts are useful if we need to rollback or restore a system.

Consistent global cuts provide a set of consistent system states.

Let us answer questions about the global properties of the system.

Note that we still don’t necessarily have the ’true’ picture of the system.

- Concurrent events may appear in an arbitrary order in a consistent run.

We have enough information to reproduce all relevant ordering.
Typically, we don’t need exact time, just consistent ordering.

We want to ensure that events that happen before others are processed first.

The space-time diagram provides a useful picture of message propagation through a system.

If we’re only concerned about consistent observations, logical clocks work fine.

Vector clocks are needed when we’re concerned about propagation of information through a system. (causal delivery)