Lecture 11: Roadmap

- Vector Clocks and Causal Delivery
- Distributed Snapshots
- Stable Predicates
- Atomic Object: Requirements
Properties of Vector Clocks

Property 1. (Strong Clock Condition.)

\[ e \rightarrow e' \equiv VC(e) < VC(e') \]

Property 2. (Simplified Strong Clock Condition.) For event \( e_i \) of \( p_i \) and event \( e_j \) of \( p_j \),

\[ e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i] \]

Property 3. (Concurrency.) For event \( e_i \) of \( p_i \) and event \( e_j \) of \( p_j \),

\[ e_i \parallel e_j \equiv VC(e_i)[i] > VC(e_j)[i] \land VC(e_j)[j] > VC(e_i)[j] \]

Property 4. Event \( e_i \) of \( p_i \) is pairwise inconsistent with event \( e_j \) of \( p_j \) iff

\[ VC(e_i)[i] < VC(e_j)[i] \lor VC(e_j)[j] < VC(e_i)[j] \]

The two disjuncts characterize the two possibilities for a cut to include one receive event without including its corresponding send event. Note that two events may be causally related, yet pairwise consistent.
Properties of Vector Clocks

Property 5. A cut defined by \((c_1, \ldots, c_n)\) is consistent iff for all \(i\) and \(j\),

\[ VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i] \]

Thus, in the frontier of a consistent cut there are not pairwise inconsistent events. It therefore suffices to check pairwise inconsistency only for events in the frontier.

Property 6. (Counting) Given event \(e_i\) of \(p_i\) and \(VC^i(e_i)\), the number of events \(e''\) such that \(e'' \rightarrow e_i\) is given by \(#(e_i)\) where \(#(e_i)\) is the sum of all entries in \(VC(e_i)\) minus 1.

Property 7. (Weak Gap Detection) Given event \(e_i\) of \(p_i\) and event \(e_j\) of \(p_j\), if for some \(k \neq j\) \(VC(e_i)[k] < VC(e_j)[k]\) then there exists an event \(e_k\) such that

\[ \neg(e_k \rightarrow e_i) \land (e_k \rightarrow e_j) \]

The property is “weak” since we cannot conclude if \(e_i \rightarrow e_k \rightarrow e_j\) holds.
Implementing Causal Delivery

Recall that our goal is to implement clocks such that

For every messages $m$ from $p_i$ to $p_k$ and $m'$ from $p_j$ to $p_k$, if $m$’s sending causally precedes $m'$, then $m$’s delivery causally precedes $m$’s.

A message $m$ from $p_i$ is deliverable when $p_0$ knows there is no undelivered $m'$ such that $\text{SND}(m') \rightarrow \text{SND}(m)$. To guarantee that there is no such $p_i$-event, $p_0$ has to guarantee that $TS(m)[i] - 1$ is the number of previous messages delivered from $p_i$.

To guarantee that there is no such $p_k$ event for $k \neq i$, we can use the weak gap-detection property, which implies that if for some $k \neq i$, $m_k$ is the last delivered $p_k$ message, then $TS(m_k)[k] < TS(m)[k]$ implies that then there exists a $p_k$ message $m'$ such that $\text{SND}(m_k) \rightarrow \text{SND}(m') \rightarrow \text{SND}(m)$.

Accordingly, if for all $k \neq i$, $TS(m)[k] \leq TS(m_k)[k]$, it is safe to deliver $m$. 
Implementing Causal Delivery

Assume that $p_0$ maintains an array $D[1 \ldots n]$ such that $D[k]$ is the $k^{th}$ coordinate in the timestamp of the last delivered $p_k$-message. Then $p_0$ delivers a $p_i$-message $m$ when:

$$D[i] = TS(m)[i] - 1$$

(1)

$$D[k] \geq TS(m)[k] \quad \forall k \neq i$$

(2)

We now have a solution to the global predicate evaluation problem, using passive observations. Since each global state the monitor constructs is consistent, it can evaluate predicates on any of the states.

If processes communicate through broadcasts, then each process can construct consistent global states. It is also possible to extend delivery rule and vector clocks to guarantee causal delivery for the case of point-to-point communication.
Beware of Hidden Channels

Thus, from the top process’s point of view,

\[ \text{pressure drop} \rightarrow \text{apply heat} \rightarrow \text{rupture} \]

while “in reality”

\[ (\text{pressure drop} \parallel \text{rupture}) \rightarrow \text{apply heat} \]
Distributed Snapshots

Let’s turn back to active monitoring, where $p_0$ requests the state information from processes. Assume that channels are FIFO. Each process will record its local state and the states of all its incoming channels. The protocol is due to Chandy and Lamport (1985).

1. Process $p_0$ starts the protocol by sending itself a “take snapshot” message.

2. When $p_i$ receives its first “take snapshot” message from $p_j$, it records its local state, $\sigma_i$ and sends “take snapshot” along all its outgoing channels. For each $k$ such that there is a $p_k$-to-$p_i$ channel, $p_i$ sets $\chi_{ki} = \lambda$.

3. Every protocol message that $p_i$ receives is then recorded in the appropriate $\chi_{ki}$, until $p_i$ receives a “take snapshot” on the channel, at which point $\chi_{ki}$ remains stable.

4. When all $\chi_{ki}$s are stable, $p_i$ sends its state and its channel information to $p_0$. 
Correctness of Protocol

Termination is guaranteed. Thus, we have to show that the global state constructed is consistent. In fact we show a stronger claim: If the snapshot algorithm was initiated when the system is in global state $\Sigma^i$, terminates when the system is in global state $\Sigma^f$, and reports global state $\Sigma^s$, then

$$\Sigma^i \rightarrow \Sigma^s \rightarrow \Sigma^f$$

where $\rightarrow$ denotes “reachable.” Thus, the reported state could have been reached in some run that leads from $\Sigma^i$ to $\Sigma^f$.

For every process $p_i$, let $e^*_i$ denotes the event when $p_i$ receives its first “take snapshot” message. Every $p_i$-event that causaly precedes $e^*_i$ is called a pre-recording event, and every $p_i$-event that causaly follows $e^*_i$ is called a post-recording event.

Assume that the “real” run that occurred is $r$. We swap consecutive global states of $r$, maintaining causality, until all the pre-recording events precede all the post-recording events, and eventually obtain a consistent run $r'$ such that $\Sigma^i \rightarrow_{r'} \Sigma^s \rightarrow_{r'} \Sigma^f$. 
From $r$ to $r'$

Consider two adjacent events $\langle e, e' \rangle$ of $r$ such that $e$ is a post-recording event and $e'$ is a pre-recording event. We show that $e \not\rightarrow e'$ so that we can safely swap the events. Assume that $e \rightarrow e'$. Then one of the two cases must obtain:

1. Both $e$ and $e'$ belong to the same process. Then, since they are post-/pre- recording events, $e' \rightarrow e$ contradicting $e \rightarrow e'$.

2. $e$ is a $\text{SND}_i$ event and $e'$ is its corresponding $\text{RCV}_j$. But, since $e$ is a post-recording event, $p_i$ had sent $p_j$ a “take snapshot” message before $e$, and, since the channel is FIFO, the message would have been received before $e'$. Thus, $e'$ is also a post-recording event.

It thus follows that $e \not\rightarrow e'$ and the two events can be safely swapped.
Stable Predicates

Consider a stable predicate, i.e., a predicate that once true remains true. For example, if the distributed snapshot protocol results in states $\Sigma^i \rightarrow \Sigma^s \rightarrow \Sigma^f$ as above, then for a stable $\Phi$ we have:

$$\Sigma^s \models \Phi \implies \Sigma^f \models \Phi$$
$$\Sigma^s \models \neg \Phi \implies \Sigma^i \models \neg \Phi$$

Examples of stable predicates are deadlock, termination, and loss of tokens.

Obviously, the distributed snapshot algorithm is a good tool for evaluating stable predicates. But for nonstable predicates, the algorithm seems useless—what good is knowing that something “may have held”?

There are, however, algorithms for detecting global predicates of the type “there exists a consistent observation of the computation where $\Phi$ holds”, and “For every consistent observation of the computation there exists a global state where $\Phi$ holds.”
Atomic Objects

Informally, an atomic register is a shared object that is accessed (possibly concurrently) by several processes. The access to the object must be atomic – i.e., be consistent with access in some “serialized” execution consistent (in a precise way) with the “real” execution.

Hardware provides us with single writer/singler reader (1W/1R) R/W atomic objects, upon which its possibly to build successively more powerful registers in a simple easy-to-verify manner. The more complex atomic objects can be used as building blocks in algorithms. (E.g., in Peterson’s we saw the need for nW/nR W/R atomic registers as well as for the simpler 1W/nR registers.)

On the abstract level, each register is specified by a set of values $V$, an initial value $v_0 \in V$, a set of invocations, a set of responses, and a function $f: \text{invocations} \times V \rightarrow \text{responses} \times V$ such that $f(i, v) = (r, v')$ describes the response $r$ obtained when $i$ is invoked when the value of the variable is $v$, and the value $v'$ of the object after the invocation. E.g., for a r/w register, $f(r, v) = (v, v)$ and $f(w(v'), v) = (\text{ack}, v')$. 
External Interface of Atomic Objects

Each object is accessed through a fixed set of input ports, each associated with some invocation, and a fixed set of output ports, each associated with some response. We assume that for every allowed invocation/response there is a corresponding input/output. For simplicity, we assume a single port matching every (invocation, response) pair. We assume that each port is also associated with a (input) stop action.

We assumed well-formedness—invocations at each port are strictly sequential, and there is a response between each two. That is, there is well-formedness on every port $i$. 
Atomicity

Very Informal Definition: Given an execution $\eta$ of the system, we say that $\eta$ satisfies the atomicity requirements if it is possible to construct an execution $\eta'$ over $\eta$’s operations that preserve the ordering of events for every port $i$ and schedules responses right after their invocations (“serializability points”) for every complete operation and some of the incomplete operations, such that causality is not violated.

More formally (but not as in Section 13.1), consider the invocations and responses of $\eta$ as a sequence $op_1, op_2, \ldots$. With every pair $(\text{inv}, \text{res})$ such that $\text{inv} = op_i$ and $\text{res} = op_j$ in the sequence, we associate the interval $(i, j)$. This interval captures, in some sense, the time frame in which the operation was executed and the response produced. We say that $(\text{inv}_1, \text{res}_1)$ associated with the interval $(i_1, j_1)$ precedes $(\text{inv}_2, \text{res}_2)$ associated with the interval $(i_2, j_2)$, and denote it by $(\text{inv}_1, \text{res}_1) \rightarrow (\text{inv}_2, \text{res}_2)$, if $j_1 < j_2$. If neither pair precedes the other, we say the they are concurrent. Incomplete invocations invoked at $op_i$ are associated with an interval $(i, \infty)$. 
Atomicity

We can now define the “serializability points” more accurately: The invocations and responses in $\eta$ are rearranged, such that each invocation is immediately followed by its response (for the complete and some of the incomplete operations) and the operation maintain their precedence relation. Thus, if in the original execution $(\text{inv}_1, \text{res}_1) \rightarrow (\text{inv}_2, \text{res}_2)$, then this is maintained in the new execution. Executions that can be rearranged this way are atomic.

Note that incomplete operations can either be ignored, or, if we choose to include them (and we must do that if they are some “write” whose results are read later by complete operations), there is no constraint by the precedence relation in terms of a “time upper bound.”
Incorporating Failures into Definitions

Well-formedness and atomicity are safety properties. To guarantee liveness, we must define the “fairness” properties. Obviously, the first one is:

**Failure-Free (FF):** in each fair execution\(^1\), that is failure-free, every invocation has a matching response.

We then have:

**Wait-Free (WF):** in each fair execution, every invocation on a non-failing port has a matching response.

\( f \) **Failure-Free:** in each fair execution with \( f \) or less faults, every invocation on a non-failing port has a matching response.

---

\(^1\) fairness here means fairness of the system in the usual sense, e.g., each locally controlled action that is always enabled is eventually taken.
An Example: Read/increment Atomic Objects

This is an example of how to implement a higher level shared object on top of simpler shared objects. In particular, how to implement a $nR/nW$ Read/Increment atomic object on top of $n$ 1W/nR r/w atomic objects. The operation allowed on the object are Read and Inc that increments it (by 1.) The object can be implemented using $n$ copies of the 1W/nR object, one at each site. Each process remembers the value of its local variable, say $v_i$.

To Read the new object $O$, every process issues a rd to each of the local r/w objects, and computes $V = \sum_{i=1}^{n} v_i$. To Inc the new object, each process performs $wt(v + 1)$ on its local r/w object, where $v$ is the old (stored) value of the object.

To see why the Read is atomic, note that the value read is in between the value when the Read was invoked and the value when its response is obtained. Since each process increments the value by 1 at time, there must be a time in this time interval when the “real” value was equal to the value returned, thus satisfying the atomicity requirements.
From Shared Variables to Atomic Objects

Consider the “architecture” for 2-process mutual exclusion:

Both $y_1$ and $y_2$ are 1W/1R R/W atomic object, and $try$ is a 2W/2R R/W atomic object.
Implementing a 1R/1W Shared Variable

**Theorem:** For every execution of the atomic object system, there is an execution of the shared memory system, such that the two executions have the same observable behaviors, and every fair execution of the shared object system is mapped into a fair execution of the atomic object execution.
Implementing a 1R/1W Shared Variable with Stop

**Theorem:** For every execution of the atomic object system, there is an execution of the shared memory system, such that the two executions have the same observable behaviors, and every fair execution of the shared object system is mapped into a fair execution of the atomic object execution.