Distributed Algorithms
Time, clocks and the ordering of events

Alberto Montresor
University of Trento, Italy

2017/05/19

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Contents

1 Modeling Distributed Executions
   • Histories
   • Happen-before
   • Global states and cuts

2 Global predicate evaluation
   • Problem definition
   • Example: deadlock detection

3 Real clocks vs logical clocks
   • Logical clocks
   • Scalar logical clocks
   • Vector logical clocks

4 Passive monitoring
   • Passive monitoring, v.1
   • Passive monitoring, v.2
   • Passive monitoring, v.3

5 Proactive monitoring
   • Snapshot protocol, v.1
   • Snapshot protocol, v.2
   • Snapshot protocol, v.3

6 Stable predicates
   • Deadlock detection
   • Deadlock detection through active monitoring
   • Deadlock detection through passive monitoring

7 Non-stable predicates
## Distributed Execution

### Definition (Distributed algorithm)
A **distributed algorithm** is a collection of distributed automata, one per process.

### Definition (Distributed execution)
The **execution** of a distributed algorithm is a sequence of **events** executed by the processes:
- **Partial execution**: a finite sequence of events
- **Infinite execution**: a infinite sequence of events

### Possible events
- \textit{send}(m, p): sends a message \textit{m} to process \textit{p}
- \textit{receive}(m): receives a message \textit{m}
- **local events** that change the local state
Histories

Definition (Local history)
The local history of process $p_i$ is a (possibly infinite) sequence of events $h_i = e_i^0 e_i^1 e_i^2 \ldots e_i^{m_i}$ (canonical enumeration)

Definition (Partial history)
The partial history up to event $e_i^k$ is denoted $h_i^k$ and is given by the prefix of the first $k$ events of $h_i$
Local histories do not specify any relative timing between events belonging to different processes.

We need a notion of ordering between events, that could help us in deciding whether:

- one event occurs before another
- they are actually concurrent
Happen-Before

**Definition (Happen-before)**

We say that an event $e$ **happens-before** an event $e'$, and write $e \rightarrow e'$, if one of the following three cases is true:

1. $\exists p_i \in \Pi : e = e^r_i, \ e' = e^s_i, \ r < s$
   
   (e and $e'$ are executed by the same process, $e$ before $e'$)

2. $e = \text{send}(m, \ast) \land e' = \text{receive}(m)$
   
   (e is the send event of a message $m$ and $e'$ is the corresponding receive event)

3. $\exists e'' : e \rightarrow e'' \rightarrow e'$
   
   (in other words, $\rightarrow$ is transitive)
Space-Time Diagram of a Distributed Computation
Meaning of Happen-Before

If \( e \to e' \), this means that we can find a series of events \( e^1 e^2 e^3 \ldots e^n \), where \( e^1 = e \) and \( e^n = e' \), such that for each pair of consecutive events \( e^i \) and \( e^{i+1} \):

1. \( e^i \) and \( e^{i+1} \) are executed on the same process, in this order
2. \( e^i = \text{send}(m,*) \) and \( e^{i+1} = \text{receive}(m) \)

Notes:

- \textit{happen-before} captures the concept of potential causal ordering
- \textit{happen-before} captures a flow of data between two events.
- Two events \( e, e' \) that are not related by the happen-before relation \( (e \not\to e' \land e' \not\to e) \) are \textit{concurrent}, and we write \( e \parallel e' \).
Prove or disprove that the $||$ relation is transitive.
Reality check

Forse non tutti sanno che...

The multi-threaded memory model of Java is based on the happen-before relation, where communication between threads is based on the acquisition and release of locks.

http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html
Global States

**Definition (Local state)**
- The **local state** of process $p_i$ after the execution of event $e_i^k$ is denoted $\sigma_i^k$.
- The local state contains all data items accessible by that process.
- Local state is completely private to the process.
- $\sigma_i^0$ is the **initial state** of process $p_i$.

**Definition (Global state)**
The **global state** of a distributed computation is an $n$-tuple of local states $\Sigma = (\sigma_1, \ldots, \sigma_n)$, one for each process.
A cut of a distributed computation is the union of \( n \) partial histories, one for each process:

\[
C = h^c_1 \cup h^c_2 \cup \ldots \cup h^c_n
\]

- A cut may be described by a tuple \((c_1, c_2, \ldots, c_n)\), identifying the frontier of the cut, i.e. the set of last events, one per process.
- Each cut \((c_1, \ldots, c_n)\) has a corresponding global state \((\sigma^{c_1}_1, \sigma^{c_2}_2, \ldots, \sigma^{c_n}_n)\).
Cuts

Alberto Montresor (UniTN)
Consistent cut

Consider cuts $C'$ and $C$ in the previous figure.

- Is it possible that cut $C$ correspond to a “real” state in the execution of a distributed algorithm?

- Is it possible that cut $C'$ correspond to a “real” state in the execution of a distributed algorithm?
**Consistent cut**

**Definition (Consistent cut)**
A cut $C$ is **consistent**, if for all events $e$ and $e'$,

\[(e \in C) \land (e' \rightarrow e) \Rightarrow e' \in C\]

**Definition (Consistent global state)**
A global state is **consistent** if the corresponding cut is consistent.

In other words:
- A consistent cut is left-closed w.r.t. the happen-before relation
- All messages that have been received must have been sent before
In the previous figures, $C$ is consistent and $C'$ is not.

In the space-time diagram, a cut $C$ is consistent if all the arrows start on the left of the cut and finish on the right of the cut.

Consistent cuts represent the concept of scalar time in distributed computation: it is possible to distinguish between a “before” and an “after”.

Predicates can be evaluated in consistent cuts, because they correspond to potential global states that could have taken place during an execution.
Contents

1 Modeling Distributed Executions
   - Histories
   - Happen-before
   - Global states and cuts

2 Global predicate evaluation
   - Problem definition
   - Example: deadlock detection

3 Real clocks vs logical clocks
   - Logical clocks
   - Scalar logical clocks
   - Vector logical clocks

4 Passive monitoring
   - Passive monitoring, v.1
   - Passive monitoring, v.2
   - Passive monitoring, v.3

5 Proactive monitoring
   - Snapshot protocol, v.1
   - Snapshot protocol, v.2
   - Snapshot protocol, v.3

6 Stable predicates
   - Deadlock detection
   - Deadlock detection through active monitoring
   - Deadlock detection through passive monitoring

7 Non-stable predicates
Introduction

Definition (Global Predicate Evaluation)
The problem of detecting whether the global state of a distributed system satisfies some predicate $\Phi$.

Motivation

- Many problems in distributed systems require a reaction when the global state of the system satisfies some condition.
  - **Monitoring**: Notify an administrator in case of failures
  - **Debugging**: Verify whether an invariant is respected or not
  - **Deadlock detection**: can the computation continue?
  - **Garbage collection**: like Java, but distributed

- Thus, the *ability to construct a global state* and evaluate a predicate over it is a core problem in distributed computing.
Examples

Garbage collection

Deadlock
Why GPE is difficult

A global state obtained through remote observations could be

- **obsolete**: represent an old state of the system.
  
  *Solution*: build the global state more frequently

- **inconsistent**: capture a global state that could never have occurred in reality
  
  *Solution*: build only consistent global states

- **incomplete**: not “capture” every moment of the system
  
  *Solution*: build all possible consistent global states
Space-Time Diagram of a Distributed Computation

Global predicate evaluation

Example: deadlock detection
Example – Deadlock detection on a multi-tier system

Processes in the previous figures use RPCs:

- Client sends a *request* for method execution; blocks.
- Server receives *request*.
- Server executes method; may invoke methods on other servers, acting as a client.
- Server sends *reply* to client
- Clients receive *reply*; unblocks.

Such a system can deadlock, as RPCs are blocking. It is important to be able to detect when a deadlock occurs.
Runs and consistent runs

**Definition (Run)**

A run of global computation is a total ordering $R$ that includes all the events in the local histories and that is consistent with each of them.

- In other words, the events of $p_i$ appear in $R$ in the same order in which they appear in $h_i$.
- A run corresponds to the notion that events in a distributed computation actually occur in a total order.
- A distributed computation may correspond to many runs.

**Definition (Consistent run)**

A run $R$ is said to be consistent if for all events $e$ and $e'$, $e \rightarrow e'$ implies that $e$ appears before $e'$ in $R$. 
Runs and consistent runs

- $e_1^1 e_2^1 e_3^1 e_4^1 e_5^2 e_6^2 e_1^2 e_2^2 e_3^2 e_4^2 e_5^3 e_3^3 \ ?$
- $e_1^1 e_2^1 e_3^1 e_4^2 e_2^2 e_3^3 e_1^4 e_4^4 e_2^3 e_3^5 e_5^6 e_1^6 \ ?$
Monitoring Distributed Computations

Assumptions:

- There is a single process $p_0$ called monitor which is responsible for evaluating $\Phi$.
- We assume that the monitor $p_0$ is distinct from the observed processes $p_1 \ldots p_n$.
- Events executed on behalf of monitoring do not alter canonical enumeration of “real” events.

In general, observed processes send notifications about local events to the monitor, which builds an observation.
Observations

**Definition (Observation)**
The sequence of events corresponding to the order in which notification messages arrive at the monitor is called an observation.

Given the asynchronous nature of our distributed system, any permutation of a run $R$ is a possible observation of it.

**Definition (Consistent observation)**
An observation is consistent if it corresponds to a consistent run.
Contents

1 Modeling Distributed Executions
   ● Histories
   ● Happen-before
   ● Global states and cuts

2 Global predicate evaluation
   ● Problem definition
   ● Example: deadlock detection

3 Real clocks vs logical clocks
   ● Logical clocks
   ● Scalar logical clocks
   ● Vector logical clocks

4 Passive monitoring
   ● Passive monitoring, v.1
   ● Passive monitoring, v.2
   ● Passive monitoring, v.3

5 Proactive monitoring
   ● Snapshot protocol, v.1
   ● Snapshot protocol, v.2
   ● Snapshot protocol, v.3

6 Stable predicates
   ● Deadlock detection
   ● Deadlock detection through active monitoring
   ● Deadlock detection through passive monitoring

7 Non-stable predicates
How to obtain consistent observations

- The happen-before relation captures the concept of potential causality

- In the “day-to-day” life, causality/concurrency are tracked using physical time
  - We use loosely synchronized watches;
  - Example: I have withdrawn money from an ATM in Trento at 13.00 on 17th May 2006, so I can prove that I’ve not withdrawn money on the same day at 13.20 in Paris

- In distributed computing systems:
  - the rate of occurrence of events is several magnitudes higher
  - event execution time is several magnitudes smaller

- If physical clocks are not precisely synchronized, the causality/concurrency relations between events may not be accurately captured
Logical clocks

Instead of using physical clocks, which are impossible to synchronize, we use logical clocks.

- Every process has a **logical clock** that is advanced using a set of rules.
- Its value is not required to have any particular relationship to any physical clock.
- Every event is assigned a timestamp, taken from the logical clock.
- The causality relation between events can be generally inferred from their timestamps.
Logical clocks

Definition (Logical clock)
A logical clock \( LC \) is a function that maps an event \( e \) from the history \( H \) of a distributed system execution to an element of a time domain \( T \):

\[
LC : H \rightarrow T
\]

Definition (Clock Condition)
\[
e \rightarrow e' \Rightarrow LC(e) < LC(e')
\]

Definition (Strong Clock Condition)
\[
e \rightarrow e' \iff LC(e) < LC(e')
\]
Scalar logical clocks

Definition (Scalar logical clocks)
- Lamport’s **scalar** logical clock is a monotonically increasing software counter
- Each process $p_i$ keeps its own logical clock $LC_i$
- The **timestamp** of event $e$ executed by process $p_i$ is denoted $LC_i(e)$
- Messages carry the **timestamp** of their *send* event
- Logical clocks are initialized to 0

Update rule
Whenever an event $e$ is executed by process $p_i$, its local logical clock is updated as follows:

$$LC_i = \begin{cases} 
LC_i + 1 & \text{If } e_i \text{ is an internal or *send* event} \\
\max\{LC_i, TS(m)\} + 1 & \text{If } e_i = \text{receive}(m) 
\end{cases}$$
Scalar logical clocks

- p1
- p2
- p3

Events:
1 2 4 5 6 7
1 5 6
1 2 4 3 5 7
p1
p2
p3
Properties

Theorem

Scalar logical clocks satisfy Clock condition, i.e.

\[ e \rightarrow e' \Rightarrow LC(e) < LC(e') \]

Proof.

This immediately follows from the update rules of the clock.
**Total ordering:** It is also possible to provide a total order for events, by sorting based on the process identifier for events with the same timestamp.
Scalar logical clocks

Theorem

Scalar logical clocks do not satisfy Strong clock condition, i.e.

\[ LC(e) < LC(e') \iff e \rightarrow e' \]
Scalar logical clocks do not satisfy Strong clock condition, i.e. 

\[ LC(e) < LC(e') \neq e \rightarrow e' \]

\[
LC(p^1_2) < LC(p^2_3), \text{ yet } p^1_2 \nrightarrow p^2_3
\]
Causal histories clocks

**Definition (Causal History)**

The causal history of an event \( e \) is the set of events that happen-before \( e \), plus \( e \) itself.

\[
\theta(e) = \{ e' \in H \mid e' \rightarrow e \} \cup \{ e \}
\]

**Theorem**

Causal histories satisfy Strong clock condition

**Proof.**

\[
\forall e \neq e' : LC(e) < LC(e') \overset{def}{\iff} \theta(e) \subset \theta(e') \iff e \in \theta(e') \iff e \rightarrow e'
\]
**Example**

**Problem:**
Causal histories tend to grow too much; they cannot be used as “timestamps” for messages.
Vector clocks

- Causal history projection: $\theta_i(e) = \theta(e) \cap h_i = h_i^{c_i}$
- $\theta(e) = \theta_1(e) \cup \theta_2(e) \cup \ldots \cup \theta_n(e) = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$
- In other words, $\theta(e)$ is a cut, which happens to be consistent.
- Cuts can be represented by their frontiers: $\theta(e) = (c_1, c_2, \ldots, c_n)$

**Definition**

The **vector clock** associated to event $e$ is a $n$-dimensional vector $VC(e)$ such that

$$VC(e)[i] = c_i \quad \text{where} \quad \theta_i(e) = h_i^{c_i}$$
Vector clocks: implementation

- Each process $p_i$ maintains a vector clock $VC_i$, initially all zeroes;
- When event $e_i$ is executed, its vector clock is updated and assumes the value of $VC(e_i)$;
- If $e_i = send(m,*)$, the timestamp of $m$ is $TS(m) = VC(e_i)$;

Update rule

When event $e_i$ is executed by process $p_i$, $VC_i$ is updated as follows:

- If $e_i$ is an internal or send event:
  \[
  VC_i[i] = VC_i[i] + 1
  \]
- If $e_i = receive(m)$:
  \[
  VC_i[j] = \max\{VC_i[j], TS(m)[j]\} \quad \forall j \neq i \\
  VC_i[i] = VC_i[i] + 1
  \]
Example
Properties of Vector clocks

“Less than” relation for Vector clocks

\[ V < V' \iff (V \neq V') \land (\forall k : 1 \leq k \leq n : V[k] \leq V'[k]) \]

Strong Clock Condition

\[ e \rightarrow e' \iff VC(e) < VC(e') \iff \theta(e) \subset \theta(e') \]

Simple Strong Clock Condition

\[ e_i \rightarrow e_j \iff VC(e_i)[i] \leq VC(e_j)[i] \]
Properties of Vector clocks

Definition (Concurrent events)

Events $e_i$ and $e_j$ are concurrent (i.e. $e_i || e_j$) if and only if:

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

In other words, event $e_i$ does not happen-before $e_j$, and $e_j$ does not happen before $e_i$.

Example: ?
Concurrent events

\[(VC(e_i)[i] > VC(e_j)[i]) \land (VC(e_j)[j] > VC(e_i)[j])\]
Example of Concurrent Events: Example: (3,1,3) and (1,2,4)
Definition (Pairwise Inconsistent)

Events $e_i$ and $e_j$ with $i \neq j$ are pairwise inconsistent if and only if

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

In other words, two events are pairwise inconsistent if they cannot belong to the frontier of the same consistent cut. The formula characterize the fact that the cut include a receive event without including a send event.

Example: ?
Pairwise inconsistent

\[(VC(e_i)[i] < VC(e_j)[i]) \lor (VC(e_j)[j] < VC(e_i)[j])\]
Example of Pairwise Inconsistent Events: Example: (3,1,3) and (1,0,2)
Properties of Vector Clocks

**Definition (Consistent Cut)**

A cut defined by \((c_1, \ldots, c_n)\) is **consistent** if and only if:

\[
\forall i, j \in [1 \ldots n]: \ VC(e_{ci}^i)[i] \geq VC(e_{cj}^j)[i]
\]

In other words, a cut is consistent if its frontier does not contain any pairwise inconsistent pair of events.
Contents

1. Modeling Distributed Executions
   - Histories
   - Happen-before
   - Global states and cuts

2. Global predicate evaluation
   - Problem definition
   - Example: deadlock detection

3. Real clocks vs logical clocks
   - Logical clocks
   - Scalar logical clocks
   - Vector logical clocks

4. Passive monitoring
   - Passive monitoring, v.1
   - Passive monitoring, v.2
   - Passive monitoring, v.3

5. Proactive monitoring
   - Snapshot protocol, v.1
   - Snapshot protocol, v.2
   - Snapshot protocol, v.3

6. Stable predicates
   - Deadlock detection
   - Deadlock detection through active monitoring
   - Deadlock detection through passive monitoring

7. Non-stable predicates
A Passive Approach to GPE

How it works

- At each (relevant) event, each process sends a notification to the monitor describing its local state.
- The monitor collects the sequence of notifications, i.e. an observation of the distributed run.

An observation taken in this way can correspond to:

- A consistent run
- A run which is not consistent
- No run at all

Can you find examples of the three cases? Can you explain why this happens?
Observations which are not runs

Problem
Observations may not correspond to a run because the notifications sent by a single process to the monitor may be delayed arbitrarily and thus arrive in any possible order.

Solution
To adopt communication channels between the processes and the monitor that guarantee that notifications are never re-ordered.
Passive monitoring

Notification ordering

**Definition (FIFO Order)**

Two notifications sent by $p_i$ to $p_0$ must be delivered in the same order in which they were sent:

$$\forall m, m': \text{send}_i(m, p_0) \rightarrow \text{send}_i(m', p_0) \Rightarrow \text{deliver}_0(m) \rightarrow \text{deliver}_0(m')$$

Uh? What is “deliver”? 

How to order notifications?

- To be ordered, each notification $m$ carries a timestamp $TS(m)$ containing “ordering” information.
- The act of providing the monitor with a notification in the desired order is called delivery; the event $deliver(m)$ is thus distinct from $receive(m)$.
- The rule describing which notifications can be delivered among those received is called delivery rule.
FIFO Order - Implementation

- Each process maintains a local sequence number incremented at each notification sent.
- The timestamp of a notification corresponds to the local sequence number of the sender at the time of sending.

**Definition (FIFO Delivery Rule)**

If the last notification delivered by $p_0$ from $p_j$ has timestamp $s$, $p_0$ may deliver “any” notification $m$ received from $p_j$ with $TS(m) = s + 1$. 
Observations which are not consistent runs

Problem
If we use FIFO order between all processes and $p_0$, all the observations taken by $p_0$ will be *runs*; but there is no guarantee that they are *consistent runs*.

Solution
To adopt communication channels that guarantee that notifications are delivered in an order that respects the happen-before relation.
Definition (Causal Order)

Two notifications sent by \( p_i \) and \( p_j \) to \( p_0 \) must be delivered following the happen-before relation:

\[
\forall m, m' : send_i(m, p_0) \rightarrow send_j(m', p_0) \Rightarrow deliver_0(m) \rightarrow deliver_0(m')
\]

Question

FIFO order among all channels...
Is it sufficient to obtain Causal delivery?
Example
• Given that each communication channels is associated with just one message, this distributed system satisfies FIFO order.

• However, it does not satisfy Causal order: the deliver of message $m'$ should be delivered after message $m$, because the send event of $m$ is potentially caused by $m'$.

• Silly example: $p_1$ sends an invitation for a party to $p_2$ and $p_3$ by SMS; $p_2$ sends a message to $p_3$ saying “Are you going to participate to the party of $p_1$?” $p_3$ gets offended because it has not been invited.
Causal delivery and consistent observations

Theorem

If $p_0$ uses a delivery rule satisfying Causal Order, then all of its observations will be consistent.

Proof.

Definition of Causal Order $\equiv$ definition of a consistent observation

Three implementations of the causal delivery rule:

- Real-time clocks
- Logical (scalar) clocks
- Vector clocks
Causal delivery and consistent observations

Passive monitoring

• Suppose, by contradiction, that the monitor obtains an observation which is not consistent.

• This means that an event $e'$ appears before $e$ in the observation, but $e \rightarrow e'$.

• This means that the notification of $e'$ has been delivered before the notification of $e$

• But this is impossible, because $send(m_e, p_0) \rightarrow send(m_{e'}, p_0)$ and by the Casual Delivery rule $deliver(m_e) \rightarrow deliver(m_{e'})$, a contradiction.
Passive monitoring with real-time

Initial assumptions

- All processes have access to a real-time clock $RC$
- Let $RC(e)$ be the real-time at which $e$ is executed
- All notifications are delivered within a time $\delta$
- The timestamp of notification $m$ corresponding to an event $e$ is $TS(m) = RC(e)$.

Definition (DR1: Real-time delivery rule)

At any time, deliver all received notifications in increasing timestamp order.

Theorem

*Observation* $O$ constructed by $p_0$ using DR1 is guaranteed to be consistent (?)
Passive monitoring with real-time

Initial assumptions

- All processes have access to a real-time clock $RC$
- Let $RC(e)$ be the real-time at which $e$ is executed
- All notifications are delivered within a time $\delta$
- The timestamp of notification $m$ corresponding to an event $e$ is $TS(m) = RC(e)$.

Definition (DR1: Real-time delivery rule)

At any time, deliver all received notifications in increasing timestamp order.

Theorem

Observation $O$ constructed by $p_0$ using DR1 is guaranteed to be consistent.
Stability of messages

Definition (Stability)
A notification $m$ received by $p_0$ is **stable at $p_0$** if no notification $m'$ with $TS(m') < TS(m)$ can be received in the future by $p_0$

Definition (DR1: Delivery rule for $RC'$)
Deliver all received notifications that are stable at $p_0$, in increasing timestamp order

Theorem

*Observation $O$ constructed by $p_0$ using DR1 is guaranteed to be consistent*
Proof

Safety: Clock Condition for $RC$

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

Note that $RC(e) < RC(e') \not\Rightarrow e \rightarrow e'$, but this rule is sufficient to obtain consistent observations, as two notifications $e \rightarrow e'$ are never delivered in the incorrect order.

Liveness: Stability

At time $t$, any message sent by time $t - \delta$ is stable.

Note that real-time clocks do not support stability; it is the maximum delay of messages that enables it.
Passive monitoring with logical clocks

Initial assumptions

- All processes have access to a logical clock $LC$; let $LC(e)$ be the logical clock at which $e$ is executed.
- The timestamp of notification $m$ corresponding to an event $e$ is $TS(m) = LC(e)$

Definition (DR2: Deliver Rule for $LC$)
Deliver all received notifications that are stable at $p_0$ in increasing timestamp order.
Passive monitoring with logical clocks

Safety: Clock Condition for $LC$

$e \rightarrow e' \Rightarrow LC(e) < LC(e')$

Note that $LC(e) < LC(e') \n e \rightarrow e'$, but this rule is sufficient to obtain consistent observations, as two events $e \rightarrow e'$ are never ordered incorrectly.
Passive monitoring with logical clocks

Liveness: Stability

We need a way to reproduce the concept of $\delta$ in an asynchronous system, otherwise no notification will be ever delivered.

Solution

- Each process communicates with $p_0$ using FIFO delivery

- When $p_0$ receives a notification from $p_i$ describing an event $e$ with timestamp $TS(e)$, it is sure that it will never receive a message from $p_i$ describing an event $e'$ with $TS(e') \leq TS(e)$

- Stability of notification $m$ at $p_0$ can be guaranteed when $p_0$ has received at least one notification from all other processes with a timestamp greater or equal than $TS(m)$
Problems of Logical Clocks

- They add unnecessary delays to observations
- They require a constant flux of notifications from all processes
Passive Monitoring with Vector Clocks

Variables maintained at $p_0$

- $M$: the set of notifications received but not yet delivered by $p_0$
- $D$: an array, initialized to 0’s, such that $D[k]$ contains $TS(m)[k]$ where $m$ is the last notification delivered by $p_0$ from process $p_k$.

When a notification is deliverable by $p_0$?

A notification $m \in M$ from process $p_j$ is deliverable as soon as $p_0$ can verify that there is no other notification $m'$ (neither in $M$ nor in the channels) such that $send(m', p_0) \rightarrow send(m, p_0)$. 
Implementing Causal Delivery with Vector Clocks

- $m \in M$: a notification sent by $p_j$ to $p_0$
- $m'$: the last notification delivered from process $p_k$, $k \neq j$

**Definition (Weak Gap Detection)**

If $TS(m')[k] < TS(m)[k]$ for some $k \neq j$, then there exists event $send_k(m'')$ such that

$$send_k(m', p_0) \rightarrow send_k(m'', p_0) \rightarrow send_j(m, p_0)$$
Implementing Causal Delivery with Vector Clocks

Two conditions to be verified to check if $m$ can be delivered:

- There is no earlier message from $p_j$ that has not been delivered yet.
  
  **Causal Delivery Rule, Part 1** $D[j] == TS(m)[j] - 1$

- $\forall k \neq j$, let $m'$ be the last message from $p_k$ delivered by $p_0$ ($D[k] = TS(m')[k]$); we must be sure that no message $m''$ from $p_k$ exists such that: $send_k(m', p_0) \rightarrow send_k(m'', p_0) \rightarrow send_j(m, p_0)$

  **Causal Delivery Rule, Part 2**: $\forall k \neq j : D[k] \geq TS(m)[k]$

  It follows from Weak Gap Detection
Example

\[
\left( D[j] == TS(m)[j] - 1 \right) \land \left( \forall k \neq j : D[k] \geq TS(m)[k] \right)
\]
Final Comments

- We have seen how to implement Causal Delivery “many-to-one”
- The same rules apply if we implement a mechanism for implementing “one-to-many” (reliable broadcast)
1 Modeling Distributed Executions
   - Histories
   - Happen-before
   - Global states and cuts

2 Global predicate evaluation
   - Problem definition
   - Example: deadlock detection

3 Real clocks vs logical clocks
   - Logical clocks
   - Scalar logical clocks
   - Vector logical clocks

4 Passive monitoring
   - Passive monitoring, v.1
   - Passive monitoring, v.2
   - Passive monitoring, v.3

5 Proactive monitoring
   - Snapshot protocol, v.1
   - Snapshot protocol, v.2
   - Snapshot protocol, v.3

6 Stable predicates
   - Deadlock detection
   - Deadlock detection through active monitoring
   - Deadlock detection through passive monitoring

7 Non-stable predicates
Snapshot Protocol

Problem

- **Goal**: To build the global state “on demand” of the monitor.
- **How**: By taking “pictures” (snapshots) of the local state when instructed
- **Challenge**: To build a consistent global state.

Applications

- Failure recovery: a global state (**checkpoint**) is periodically saved; to recover from a failure, the system is restored to the last saved checkpoint.
- Distributed garbage collection
- Monitoring distributed events (e.g., industrial process control)
Chandy-Lamport Snapshot Algorithm

- This particular protocol enables to reason about “channel states”
- GPE can be delayed with respect to passive monitoring
- Assumption: processes communicate through FIFO channels
Snapshot Protocol

Channel State

- For each channel between $p_i$ and $p_j$
  - $x_{i,j}$ contains the messages sent by $p_i$ but not received yet by $p_j$
  - $x_{j,i}$ contains the messages sent by $p_j$ but not received yet by $p_i$

- Note: channel state can be obtained by storing appropriate information in the local state, but it is complicated.

Recorded information

Each process $p_i$ will record its local state $\sigma_i$ and the content of its incoming channels $x_{j,i}$.
Snapshot Protocol

Assumptions:

- Communication channels satisfy FIFO order

Assumptions to be relaxed later:

- Access to a real time clock $RC$;
- Message delays are bounded by some known value $\delta$;
- Relative process speeds are bounded;
- Message $m$ is tagged with a timestamp $TS(m) = RC(e)$, where $e = send(m)$.
Snapshot Protocol, v.1

1. $p_0$ chooses a time $t_s$ far enough in the future that a message containing $t_s$ sent by $p_0$ will be received by all processes by $t_s$:

   $$t_s = now() + \delta$$

2. $p_0$ sends a message "take a snapshot at $t_s" to all processes in $\Pi$

3. When $RC_i = t_s$ do:
   1. $p_i$ records its local state
   2. $p_i$ sends an empty message to all processes in $\Pi$
   3. $p_i$ starts to record all messages received over each incoming channel

4. When $p_i$ receives a message $m$ from $j$ such that $TS(m) \geq t_s$, it stops recording messages for incoming channel $j$

5. Each $p_i$ sends its recorded local state and channel states to $p_0$
Snapshot Protocol, v.1

Liveness The empty messages at 3.2 guarantee liveness.

Safety This protocol constructs a consistent global state; actually, this global state did in fact occur. Formally:

Let $C_s$ be the cut associated to the constructed global state;

$$
(e \in C_s) \land (e' \rightarrow e) \implies 
(e \in C_s) \land (RC(e') < RC(e)) \implies
$$

C.C. : $e' \rightarrow e \implies RC(e') < RC(e)$

$(e' \in C_s) \quad C_s$ Def. : $e \in C_s \Leftrightarrow RC(e) < t_s$

Can we use logical clocks instead of a real time clock?
From real time clocks to logical clocks:

- The construct “when \( LC = t_s \) do statement” makes no sense
  - \( LC \)'s are not continuous (e.g., they can jump from \( t_s - 1 \) to \( t_s + 1 \))
  - When \( LC = t_s \), the event that caused the clock update is already done.

- Solution: before \( p_i \) executes an event \( e \):
  - If \( e \) is an internal or send event, and \( LC = t_s - 2 \):
    - \( p_i \) executes \( e \)
    - \( p_i \) executes \( \text{statement} \)
  - If \( e = \text{receive}(m) \land TS(m) \geq t_s \) and \( LC < t_s - 1 \):
    - \( p_i \) puts \( LC = t_s - 1 \)
    - \( p_i \) executes \( \text{statement} \)
    - \( p_i \) executes \( e \)
Snapshot Protocol, v.2

From real time clocks to logical clocks:

- We assume that we can found a logical time $t_s$ large enough that a message containing it sent by $p_0$ will be received by all other processes before $t_s$
- Impossible in an asynchronous distributed systems
- We will relax this assumption later
Snapshot Protocol, v.2

1. $p_0$ chooses a logical time $t_s$ large enough
2. $p_0$ sends a message “take a snapshot at $t_s$” to all processes in $\Pi$
3. $p_0$ sets its logical clock to $t_s$;
4. When $LC_i = t_s$ do:
   1. $p_i$ records its local state;
   2. $p_i$ sends an empty message to all processes in $\Pi$;
   3. $p_i$ starts to record all messages received over each incoming channel.
5. When $p_i$ receives a message $m$ from $j$ such that $TS(m) \geq t_s$, it stops recording messages for incoming channel $j$
6. Each $p_i$ sends its recorded local state and channel states to $p_0$. 
We now remove the need for $t_s$:

- A process may receive an ”empty message” from a node before the “take snapshot at $t_s$” is actually received.
- In other words, it may be already aware of the snapshot protocol.
- We remove the “at $t_s$” and we use SNAPSHOT messages instead of empty ones.
- We can now remove logical clocks completely, as messages are not timestamped any more.
Snapshot Protocol, v.3

1. $p_0$ sends a message SNAPSHOT to itself;

2. when $p_i$ receives SNAPSHOT for the first time:
   - let $p_j$ be the sender of this message
   - $p_i$ records its local state $\sigma_i$;
   - sends SNAPSHOT to all processes in $\Pi$;
   - $x_{k,i} \leftarrow \emptyset \quad \forall k \neq i$;

3. when $p_i$ receives message $m \neq$ SNAPSHOT from $p_k$, $k \neq j$
   - $x_{k,i} \leftarrow x_{k,i} \cup \{m\}$

4. when $p_i$ receives a SNAPSHOT from $p_k$, $k \neq j$ beyond the first time:
   - $p_i$ stops recording messages in $x_{k,i}$;

5. when $p_i$ has received a SNAPSHOT from all processes
   - $p_i$ then sends its recorded local state and the channel states to $p_0$. 
Example

Proactive monitoring
Snapshot protocol, v.3

Normal messages
Snapshots
Recorded states
State p₁
State p₂
Incoming channel p₁

Snapshots
Recorded states
Normal messages
State p₁
State p₂
Incoming channel p₁

Alberto Montresor (UniTN)
Proof of correctness

**Theorem**

*A global state built using the Chandy-Lamport snapshot algorithm is consistent.*
Proof of correctness

Theorem

A global state built using the Chandy-Lamport snapshot algorithm is consistent.

In order for the global state to be inconsistent, there should be a receive event included in the global state for which the corresponding send event is not included. Let $e_i = \text{receive}_i(m)$ and let $e_j = \text{send}_j(m)$.

In order for $e_j$ to not belong to the global state, process $p_j$ must have received the first SNAPSHOT message before $e_j$. Thus, it has sent a message to $p_i$ containing a SNAPSHOT message to $p_i$ before sending $m$. For the FIFO property of the channel, $p_i$ must have received the SNAPSHOT message before $e_i$. So, it must have stored the local state before receiving $m$, a contradiction.
Stable Predicates

Problem
- Let $\Sigma$ be a global state built by one of the presented methods
- It represents a state of the past, potentially with no bearing to the present
- Does it make sense to evaluate predicate $\Phi$ on it?

A special case: stable predicates
Many systems properties have the characteristic that once they become true, they remain true.
- Deadlock
- Garbage collection
- Termination
A distributed computation may have many runs

Definition (Leads-to relation)

- A consistent run $R = e^1 e^2 \ldots$ results in a sequence of consistent global states $\Sigma^0 \Sigma^1 \Sigma^2 \ldots$, where $\Sigma^0$ denotes the initial global state.

- We say that a global state $\Sigma$ leads to to a global state $\Sigma'$, denoted $\Sigma \leadsto_R \Sigma'$ in a consistent run $R$ if:
  - $R$ results in a sequence of global states $\Sigma^0 \Sigma^1 \Sigma^2 \ldots$;
  - $\Sigma = \Sigma^i$, $\Sigma' = \Sigma^j$, $i < j$.

- We write $\Sigma \leadsto \Sigma'$ if there is a run $R$ such that $\Sigma \leadsto_R \Sigma'$. 
A distributed computation may have many runs

Definition (Lattice)

- The set of all consistent global states of a computation along with the leads-to relation defines a lattice;
- $n$ orthogonal axis, one per process;
- $\Sigma^{k_1 \ldots k_n}$ shorthand for the global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$;
- The level of $\Sigma^{k_1 \ldots k_n}$ is equal to $k_1 + \cdots + k_n$.
- A path in the lattice is a sequence of global states of increasing levels that corresponds to a consistent run.
Figure 3. A Distributed Computation and the Lattice of its Global States
Consider a global state construction protocol:

- Let $\Sigma^a$ be the global state in which the protocol is initiated;
- Let $\Sigma^f$ be the global state in which the protocol terminates;
- Let $\Sigma^s$ be the global state constructed by the protocol

Since $\Sigma^a \leadsto \Sigma^s \leadsto \Sigma^f$, if $\Phi$ is stable, then:

\[
\begin{align*}
\Phi(\Sigma^s) &= \text{true} & \Rightarrow & & \Phi(\Sigma^f) &= \text{true} \\
\Phi(\Sigma^s) &= \text{false} & \Rightarrow & & \Phi(\Sigma^a) &= \text{false}
\end{align*}
\]
Deadlock Detection

Code

- Server code
- Server code, modified for snapshot protocol
- Monitor code for snapshot protocol
- Server code, modified for passive protocol
- Monitor code for passive protocol

Notes

- No need to store channel state in this case
Server code

Process $p_i$

```
Queue pending ← new Queue()
boolean working ← false
while true do
    while working or pending.size() = 0 do
        receive $\langle m, p_j \rangle$
        if $m$.type = REQUEST then
            pending.enqueue($\langle m, p_j \rangle$)
        else if $m$.type = RESPONSE then
            $\langle m', p_k \rangle$ ← nextState($m, p_j$)
            working ← ($m'.$type = REQUEST)
            send $m'$ to $p_k$
    while not working and pending.size() > 0 do
        $\langle m, p_j \rangle$ ← pending.dequeue()
        $\langle m', p_k \rangle$ ← nextState($m, p_j$)
        working ← ($m'.$type = REQUEST)
        send $m'$ to $p_k$
```
Deadlock Detection through Snapshot

Approach
- All channels are based on FIFO delivery
- Add code to deal with Snapshot messages

Pros and Cons
- Generates overhead only when deadlock is suspected
- Introduces a delay between deadlock and detection
Server code, modified for active monitoring (1)

Process $p_i$

Queue pending $\leftarrow$ new Queue()

boolean working $\leftarrow$ false

boolean[] blocking $\leftarrow$ \{false, ..., false\}

while true do

while working or pending.size() = 0 do

receive $\langle m, p_j \rangle$

if $m$.type = REQUEST then

    blocking[$j$] $\leftarrow$ true
    pending.enqueue($\langle m, p_j \rangle$)

else if $m$.type = RESPONSE then

    $\langle m', p_k \rangle$ $\leftarrow$ nextState($m, p_j$)
    working $\leftarrow$ ($m'$.type = REQUEST)
    send $m'$ to $p_k$
    if $m'$.type = RESPONSE then
        blocking[$k$] $\leftarrow$ false

else

    working $\leftarrow$ false
Server code, modified for active monitoring (2)

Process $p_i$

```plaintext
else if $m.type = \text{SNAPSHOT}$ then
  if $s = 0$ then
    send $\langle\text{SNAPSHOT, blocking}\rangle$ to $p_0$
    send $\langle\text{SNAPSHOT}\rangle$ to $\Pi - \{p_i\}$
    $s \leftarrow (s + 1) \mod n$
  
while not $working$ and $\text{pending.size()} > 0$ do
  $\langle m, p_j \rangle \leftarrow \text{pending.dequeue()}$
  $\langle m', p_k \rangle \leftarrow \text{nextState}(m, p_j)$
  $working \leftarrow (m'.type = \text{REQUEST})$
  send $m'$ to $p_k$
  if $m'.type = \text{RESPONSE}$ then
    $\text{blocking}[k] \leftarrow \text{false}$
```

Alberto Montresor (UniTN)
Monitor code, for active monitoring

Process $p_0$

```plaintext
boolean[][] wfg ← new boolean[1...n][1...n]
while true do
    { Wait until deadlock is suspected }
    send ⟨SNAPSHOT⟩ to Π
    for $k ← 1$ to $n$ do
        receive ⟨$m, j$⟩
        $wfg[j] ← m.data$
    if there is a cycle in $wfg$ then
        { the system is deadlocked }
```
Deadlock detection through passive monitoring

Approach

- Sends a message to $p_0$ for each relevant event
- Communication with $p_0$ based on causal delivery

Pros and Cons

- Simpler approach, but complexity is hidden by the causal delivery mechanism
- Latency limited to message delays
- Higher overhead
Server code, modified for passive monitoring (1)

Process $p_i$

```
QUEUE pending ← new Queue()
boolean working ← false
while true do
    while working or pending.size() = 0 do
        receive $\langle m, p_j \rangle$
        if $m$.type = REQUEST then
            send $\langle$REQUESTED, $j$, $i$\rangle$ to $p_0$
            pending.enqueue($\langle m, p_j \rangle$)
        else if $m$.type = RESPONSE then
            $\langle m', p_k \rangle$ ← nextState($m$, $p_j$)
            working ← ($m'$.type = REQUEST)
            send $m'$ to $p_k$
            if $m'$.type = RESPONSE then
                send $\langle$RESPONDED, $i$, $k$\rangle$ to $p_0$
```
Server code, modified for passive monitoring (2)

Process $p_i$

\[
\textbf{while not} \ \text{working and} \ \text{pending.size()} > 0 \ \textbf{do}
\]

\[
\langle m, p_j \rangle \leftarrow \text{pending.dequeue}()
\]

\[
\langle m', p_k \rangle \leftarrow \text{nextState}(m, p_j)
\]

\[
\text{working} \leftarrow (m'.\text{type} = \text{REQUEST})
\]

\[
\text{send} \ m' \ \text{to} \ p_k
\]

\[
\text{if} \ m'.\text{type} = \text{RESPONSE} \ \textbf{then}
\]

\[
\text{send} \ \langle \text{RESPONDED}, i, k \rangle \ \textbf{to} \ p_0
\]
Monitor code, for passive monitoring

**Process $p_0$**

```plaintext
boolean[ ][ ] wfg ← new boolean[1 . . . n][1 . . . n]
while true do
    receive ⟨m, p_j⟩
    if m.type = responded then
        wfg[m.from, m.to] ← false
    else
        wfg[m.from, m.to] ← true
    if there is a cycle in wfg then
        { the system is deadlocked }
```
Contents

1 Modeling Distributed Executions
   • Histories
   • Happen-before
   • Global states and cuts

2 Global predicate evaluation
   • Problem definition
   • Example: deadlock detection

3 Real clocks vs logical clocks
   • Logical clocks
   • Scalar logical clocks
   • Vector logical clocks

4 Passive monitoring
   • Passive monitoring, v.1
   • Passive monitoring, v.2
   • Passive monitoring, v.3

5 Proactive monitoring
   • Snapshot protocol, v.1
   • Snapshot protocol, v.2
   • Snapshot protocol, v.3

6 Stable predicates
   • Deadlock detection
   • Deadlock detection through active monitoring
   • Deadlock detection through passive monitoring

7 Non-stable predicates
Non-stable predicates

Problems of non-stable predicates

- The condition encoded by the predicate may not persist long enough for it to be true when the predicate is evaluated.
- If a predicate $\Phi$ is found to be true by the monitor, we do not know whether $\Phi$ *ever* held during the *actual* run.

Conclusions

- Evaluating a non-stable predicate over a single computation makes no sense.
- The evaluation must be extended to the entire lattice of the computation.
Non-stable predicates

Predicates over entire computations

It is possible to evaluate a predicate over an entire computation using an observation obtained by a passive monitor.

- **Possibly(Φ)**: There exists a consistent observation $O$ of the computation such that $Φ$ holds in a global state of $O$.
- **Definitely(Φ)**: For every consistent observation $O$ of the computation, there exists a global state of $O$ in which $Φ$ holds.

**Example (Debugging)**

If **Possibly(Φ)** is true, and $Φ$ identifies some erroneous state of the computation, than there is a bug, even if it is not observed during an actual run.
Example: Possibly\((y - x) = 2\), Definitely\((x = y)\)

\[
\begin{align*}
\Sigma^{00} & \quad \Sigma^{10} & \quad \Sigma^{01} \\
\Sigma^{11} & \quad \Sigma^{02} \\
\Sigma^{21} & \quad \Sigma^{12} & \quad \Sigma^{03} \\
\Sigma^{31} & \quad \Sigma^{22} & \quad \Sigma^{13} \\
\Sigma^{41} & \quad \Sigma^{32} & \quad \Sigma^{23} & \quad \Sigma^{14} \\
\Sigma^{51} & \quad \Sigma^{42} & \quad \Sigma^{33} & \quad \Sigma^{24} & \quad \Sigma^{15} \\
\Sigma^{61} & \quad \Sigma^{52} & \quad \Sigma^{43} & \quad \Sigma^{34} & \quad \Sigma^{25} & \quad \Sigma^{16} \\
\Sigma^{62} & \quad \Sigma^{53} & \quad \Sigma^{44} & \quad \Sigma^{35} & \quad \Sigma^{26} & \quad \Sigma^{17} \\
\Sigma^{63} & \quad \Sigma^{54} & \quad \Sigma^{45} & \quad \Sigma^{36} & \quad \Sigma^{27} & \quad \Sigma^{18} \\
\Sigma^{64} & \quad \Sigma^{55} & \quad \Sigma^{46} & \quad \Sigma^{37} & \quad \Sigma^{28} & \quad \Sigma^{19} \\
\Sigma^{65} & \quad \Sigma^{56} & \quad \Sigma^{47} & \quad \Sigma^{38} & \quad \Sigma^{29} & \quad \Sigma^{110} \\
\end{align*}
\]

\(y - x) = 2

\(x = y\)

Initially \(x = 0\) and \(y = 10\)
Non-stable predicates

Predicates over entire computations

Theorem
Possibly and Definitely are not duals:

\[ \neg \text{Possibly}(\Phi) \nleftrightarrow \text{Definitely}(\neg \Phi) \]
\[ \neg \text{Definitely}(\Phi) \nleftrightarrow \text{Possibly}(\neg \Phi) \]

Example
Possibly \((x \neq y)\), Definitely \((x = y)\)
Algorithms for detecting Possibly and Definitely

- We use the passive approach in which processes send notifications of events relevant to $\Phi$ to the monitor $p_0$;
- Events are tagged with vector clocks;
- The monitor collects all the events and builds the lattice of global states.

**How?**

- To detect $\text{Possibly}(\Phi)$: if there exists one global state in which $\Phi$ is true, then return $\text{true}$, otherwise $\text{false}$.
- To detect $\text{Definitely}(\Phi)$: mark nodes where $\Phi$ is true with a value 1, the other nodes with value 0. If the cost of the shortest path between the initial state and the final state is larger than 0, return $\text{true}$, otherwise $\text{false}$.
Algorithms for detecting **Possibly** and **Definitely**

**Problems**

- The number of states grows exponentially with the number of total events.
- Techniques can be used to reduce the number of events
  - Only those relevant to $\Phi$
  - Forcing periodic synchronization
  - Reducing the complexity of predicates (conjunction of local predicates)
O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms.
Reality Check: Interesting links

- Clocks are bad, or welcome to distributed systems
- Why vector clocks are easy
- Why vector clocks are hard
- Why Cassandra doesn’t need vector clocks