Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms

From chapter 4 of
Distributed Systems, 2nd Ed.
Sape Mullender, editor
Addison Wesley
Consistent Global States of Distributed Systems

- Asynchronous distributed systems
- Consistent global states
- Vector clocks
- Causal delivery
- Distributed snapshots
- Global predicates
Asynchronous Distributed Systems

• Components
  – collection of sequential processes
  – network communication channels
  – message exchange

• Assumptions
  – no bounds on relative speeds of processes
  – no bounds on message delays
  – reliable/unreliable and FIFO/non-FIFO channels
Asynchronous Distributed Systems

• Distributed Computations
  – Describes the execution by a collection of processes
• Process model
  – process state = sequence of event (history)
    • send(m) \( h_i = e_1^i \land e_2^i \land \cdots \land e_k^i \), execute
  – local history
    • \( K \) is the \( k \)th event
    \[ H = \bigwedge_{i} h_i \] (하나의 process의 history)
  – global history
    • Containing all of events
Causal Ordering of Events
• “happens before” relation (→), cause-and-effect’

1. If \( e_i^k, e_i^l \in h_i \) and \( k < l \), then \( e_i^k \rightarrow e_i^l \)

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

3. If \( e \rightarrow e' \) and \( e' \rightarrow e'' \), then \( e \rightarrow e'' \)

• Concurrent event
  \( e \parallel e' \equiv (e \rightarrow e') \land (e' \rightarrow e) \)

• A distributed computation
  – A partially ordered set (p.o.set) defined by the pair \((H, \rightarrow)\)
Message delivery: FIFO but not Causal

- 5을 받을 때 6이 올 것을 알고 기다렸다 6을 먼저 delivery를 해야 Causal delivery이다.
An example of Distributed Computation

If a path can be traced from one event to the other proceeding left-to-right along the horizontal lines and in the sense of the arrows, they are related, otherwise they are concurrent.

In the figure, $e_{21} \rightarrow e_{34}$, but $e_{22} \parallel e_{34}$.
Consistent Global State

Global state: n-tuple of local states

\[ \sum = (\sigma_1, \sigma_2, \Lambda, \sigma_n) \]

Cut: subset \( C \) of its global history \( H \)

Run: total ordering \( R \) of all events in \( H \) that is consistent with each local history

Consistent Cut: \( e \rightarrow e' \), \( e' \)이 포함되면 \( e \)도 포함

Consistent Run \( e \rightarrow e' \Rightarrow e \) appears before \( e' \) in \( R \)

Lattice of its global states

\( X_{k_1 \ldots k_n} \) be a shorthand for the global state

\( k_1 + \ldots + k_n \) be its level
Lattice of Global States

이런 것을 관찰해서는 안 된다.

X₃₀은 있을 수 없다. P1의 두 번째 event는 p2에게 msg를 받은 event인데, p2는 아무것도 하지 않은 그런 상태는 있을 수 없다.

X₁₅도 마찬가지다.
Cuts of Distributed Computation

C and c”” are consistent cuts, but c’ is not a consistent cut
Observing only **Consistent** Global States

- **Monitoring**
  - \( P_0 \) is monitoring other processes’ progress.
    - \( P_1, P_2, P_3, \ldots \)

- **Snapshot**
  - \( P_0 \) wants to record the global states from time to time.
    - by broadcasting “**take a snapshot and report**”.
Consistent Monitoring

- $P_0$ is monitoring other processes.
  - $P_1, P_2, P_3, \ldots$
- Other processes send a message to $p_0$ whenever they execute an event.
- Messages can be delayed.....
- Messages should not be delivered as they are received.
  - delivery should be causal as well as FIFO.
- How can we make sure (reorder the messages) so that $p_0$ only records a consistent run?
Message delivery: FIFO but not Causal

- 5를 받을 때 6이 올 것을 알고 기다렸다 6을 먼저 delivery를 해야 Causal delivery이다.
Observing only Consistent Global States

• Deliver the received message only when it is certain that delivering it would not violate the consistency!

• How can we tell?
  – If we can be sure that the message is the oldest.

• 3 approaches
  – Use synchronized Real-time clock
  – Use logical clock
  – Use vector clock
Observing Consistent Global States using synchronized Real-time clock
Synchronized Real time clock (RC)

• Clocks should be synchronized
  – within some bound $w$.
• Message delay should be bounded by $d$.
• Messages should be delivered FIFO.
• Then, $\delta$ is the maximum message delay
  – $\delta = w + d$.
• At time $t$, we can be sure that no more message would arrive with timestamp less than $t-\delta$. 
Observing Consistent Global States using logical clock (LC)
Logical Clock

- Logical Clock
  \[ LC(e_i) := \begin{cases} 
  LC + 1 & \text{on each internal/send event} \\
  \max\{LC, TS(m)\} + 1 & \text{on receive event} 
  \end{cases} \]

- LC lacks Gap-Detection property:
  - If \( LC(e) < LC(e') \), determine whether some other event \( e'' \) exists such that \( LC(e) < LC(e'') < LC(e') \)

- Strong Clock Condition: 
  \[ e \to e' \equiv TC(e) < TC(e') \]

- RC, LC? Weak Clock

\[ e \to e' \implies RC(e) < RC(e'), \ LC(e) < LC(e') \]
\[ RC(e) < RC(e'), \ LC(e) < LC(e') \implies e \to e' \]
Logical clocks

- P1에서 1과 4, P2에서 1과 5, P3에서 5과 7사이에 다른 event가 있는지 알 수가 없다.
Causal Delivery

• How to order messages
  – to prevent inconsistent global states from being observed?

• When to deliver an incoming msg?
  – deliver stable msgs in timestamp order
  • stable : known last msg to deliver
  – how to determine if it’s stable?
  • at least one msg received from all other processes with a timestamp greater than TS(m)
Observing Consistent Global States using vector clock (VC)
Causal History

- C.H. of e in $(H, \rightarrow)$: $\theta(e) = \{e' \in H | e' \rightarrow e\} \cup \{e\}$
  - Smallest consistent cut including e
  - When used as clock values, strong clock condition is met:
    $e \rightarrow e' \equiv \theta(e) \subset \theta(e')$
- If we send CH along with each msg, we know causal order of them!
Causal history of $e_{14}$

$\Theta(e_{14}) = \{e_{11}, e_{12}, e_{13}, e_{14}, e_{21}, e_{31}, e_{32}, e_{33}\}$
Vector Clocks

\[ VC(e)[i] = k, \text{iff} \theta_i(e) = h_i^k \]

- **VC update rule**
  \[
  \begin{align*}
  VC(e_i)[i] &:= VC[i] + 1 \quad \text{if } e_i \text{ is internal / send event} \\
  VC(e_i) &:= \max\{VC, TS(m)\} \\
  VC(e_i)[i] &:= VC[i] + 1 \quad \text{if } e_i \text{ is receive}
  \end{align*}
  \]

- **VC interpretation**
  \[ VC(e_i)[j] = \text{number of events of } p_j \text{ that causally precede event } e_i \text{ of } p_j \]

- **VC < relation**
  \[ V < V' \equiv (V \neq V') \land (\forall k: 1 \leq k \leq n: V[k] \leq V'[k]) \]
Vector clocks

![Diagram of Vector clocks]

울 msg가 내가 알고 있는 것보다 많이 알고 있으면 그 것을 취한다.
Vector Clock Properties

- **Strong clock** \( e \preceq e' \iff VC(e) < VC(e') \)
- **Simple strong clock** \( e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i] \)
- **Concurrent events** (각자의 행동에 대해 자신이 더 많이 알고 있을 때)
  \( e_i || e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \land (VC(e_j) > VC(e_i)[j]) \)
- **Pairwise inconsistent events** (자기가 속한 프로세스의 이벤트에 대해 상대방이 더 많이 알고 있을 때)

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

- **Consistent Cut** A cut \( \{c_1, c_2, \ldots, c_n\} \) is consistent if

\[ \forall i, j: 1 \leq i \leq n, 1 \leq j \leq n: VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i] \]

Distributed Computing Systems 35
Concurrent events: (1,0,0) vs (0,1,0), (4,1,3) vs (1,0,5)
Pairwise inconsistent events: (4,1,3) vs (5,1,6), (3,1,3) vs (1,0,2)
Consistent cut: (5,1,3) (1,2,4) (1,0,5)

p1, p2, p3 모두 자신에 대해 가장 잘 알고 있다

Distributed Computing Systems
Vector Clock Properties

- **Counting:** \( #(e_j) = (\sum_{j=1}^{n} VC(e_i)[j]) - 1 \)
  - \( #(e_i) \) denotes the numbers of events that causally precede \( e_i \) in the entire computation

- **Weak Gap detection**

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

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

(not implies \( e_i \rightarrow e_k \rightarrow e_j \)) $\rightarrow$ strong
Causal Delivery using VC

- A message can be delivered if $p_0$ can be sure that there is no message causally precedes it.
- There are other messages not yet delivered (other events $p_0$ does not know yet), but they are at least concurrent with it.
- $P_0$ maintains a cut and the cut should be consistent at all times !!!
  - FIFO delivery from the same channel.
  - No receiving a message not yet sent.
Causal Delivery using VC

- Deliver msg_m from process p_j as soon as both the following conditions are met

\[ D[j] = TS(m)[j] - 1 \]
\[ D[k] \geq TS(m)[k], \forall k \neq j \]
\[ D[i] \text{ contains } TS(m_i)[i] \text{ where } m_i \text{ is the last message delivered from } p_i \]
- D[-] is the cut p_0 maintains.
- after delivery of m, set D[j] = TS(m)[j]
Causal delivery using vector clocks

- **P**₁에 대한 **event** 는 **p**₁에게서 알아야지 (1,1)을 **deliver** 하게되면 **p**₂에게서 알게되는 꼴이 된다. 따라서 (1,0)가 온때까지 기다려야…
  - 하나씩 **increment**한 것을 받아야 한다.갑자기 **jump**하는 것을 받으면 **wait** 해야 한다.
- **m’** is delayed until message **m** has been received and delivered.
- **m’’** can be delivered as soon as it is received.
Distributed Snapshots

• SP1) with Realtime clock RC
  – P0 : ‘take snapshot at tss’ to all processes
  – each Pi takes snapshot at tss and start recording incoming msgs
  – when receiving m s.t. TS(m)>tss, stop recording of that channel and report recorded msgs
• No lost msgs, no inconsistent SS
Distributed Snapshots

SP2) Replace RC with LC in SP1

- P0: ‘at w do S’ to all processes
- LC = w, Pi takes SS and start recording
- As soon as receiving TS >= w from other process, stop recording and report that channel

RC, LC 모두 msg가 제 순서로 가는 것을 어느 정도 보장한다는 가정
Distributed Snapshots

• SP3) Eliminating role of clock
  – P0 sends ‘take SS’ to himself
  – Pi : when receiving first ‘take SS’ from pf, take SS and relays ‘take SS’ to everybody, \(X_{f,i}\) is set to empty, and starts recording.
  – When Pi receives ‘take SS’ beyond the first time from j, stops recording channel from j (\(X_{j,i}\))
Properties of Snapshots

• An Example, think this:
  – (2,3) is observed global state
  – An actual run might not get passed (2,3)
  – Prerecording \((e_i \rightarrow e_i^*)\)/ Postrecording
  – Pre/Post recording events can be swapped
    • Eventually, observed state can be converted to show it was ‘possible’ to happen
  – SP3 guarantees only consistent global states are observed
An Example of SS Protocol 3

Ch₁₀ = {}  
Ch₂₀ = {}

P₂가 SS잡기 전에 올 MSG네가 기록해야 한다.

P₁에게 첫 번째 msg

두 번째 msg 넣어서 여기까지 기록했다.

SS from P₁을 받은 후, m₂를 받았으므로, 기록할 필요 없다.

Ch₂₁ = {m₁}

Ch₁₂ = {}
SS Protocol 3

P0
- takes snapshot (2)
- Ch10 = {} (2-5)
- Ch20 = {} (2-7)
- take a snapshot (1)

P1
- takes snapshot (4)
- take a snapshot (3)
- Ch01

P2
- takes snapshot (6)
- take a snapshot (5)
- Ch02 = {} (6-8)

Ch21 = {} (4-7)

나에게 SS를 잡으라고 보낸 P에게서 온 MSG는 기록할 필요 없다.
Real Computation vs. Observed State

In real computation, $X_{23}$ did not happen!

$X_{23}$ is still a consistent global state!!
Global Predicates

• Stable predicates
  – once become true, they remain true during rest of computation (eg. deadlock)

• Nonstable predicates
  – eg. Q length reached n
    – Possibly(Φ) : May happen for some run
    – Definitely(Φ) : Must happen for every run
  – Detecting Definitely() is harder
Homework II

• Due 3/21
  – We learned how the monitoring process can only record consistent run by using vector clock.
  – If we want to make sure all the communicating processes only deliver messages causally, what should be done?