Technical University of Cluj-Napoca
Computer Science Department

Distributed Systems

- Master in CS -

Review

C1 - Distributed Computation Model
C2 - Global States and Snapshots
C3 - Time and Causality

Fall 2017

Technical University of Cluj-Napoca
Computer Science Department

Distributed Systems

- Master in CS -

Review

- C1 -

Distributed Computation Model

Ioan Salomie

Fall 2017

Distributed Computation Model
Contents

- Overview
- Distributed Program
- Process Execution
- Time-Space Diagrams
- Distributed Application Execution
- Dependent and Independent Events
- Concurrent Events
- Physical and Logical Concurrency
- Network Models

Distributed System
Overview

- Set of autonomous processors connected by a communication network
  - Communication network
    - » Facilitates information exchange among processors
    - » Communication delay is finite but unpredictable
    - » May deliver messages out of order,
    - » Messages may be lost, corrupted/confused, or duplicated due to timeout and retransmission
    - » Communication links may go down.
  - The processors
    - » No common global memory and
    - » Communicate only by passing messages over the communication network
    - » May fail

Distributed System
Overview (cont.)

- No physical global clock in the system
- System model
  - Directed graph
  - Vertices - processes
  - Edges - unidirectional communication channels.
- Distributed application
  - Running collection of processes over the distributed system

Distributed System
Main Features

- Processes interact for a common goal
- Processes exchange information
- Concurrency
- Coordination problem
- No global clock
- Independently fail
- Graceful degradation

Distributed System
False Assumptions

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

Distributed Program (1)

- Distributed program
  - Consists of a set of **n** asynchronous processes:

    $p_1, p_2, \cdots p_i, \cdots p_n$
  - Inter-process communication: message passing over the communication network
- Assumptions
  - Each process is running on a different processor
  - Processes do not share a global memory
  - Processors communicate only by passing messages
  - Communication delay is finite and unpredictable
  - No global clock
- Process execution is asynchronous
  - A process may execute an action spontaneously
- Message transfer is asynchronous
  - A process sending a message does not wait for the delivery of the message to be completed

Distributed Program (2)

- Global state of a distributed computation
  - States of the processes and
  - States of the communication channels.
- The state of a process
  - The state of its local memory, registers, threads, time out timers
- The state of a channel
  - The set of messages in transit in the channel
- Notations
  - $\mathbf{C_{ij}}$ - the channel from process $\mathbf{p_i}$ to $\mathbf{p_j}$
  - $\mathbf{m_{ij}}$ denote a message sent by $\mathbf{p_i}$ to $\mathbf{p_j}$

Process Execution (1)

- Process execution
  - A sequential execution of its atomic actions
- Process atomic actions
  - Internal events
  - Message send events
  - Message receive events
- Events
  - Occurring the events
    - => State changes of processes and channels
    - => Transitions in the global system state
  - Send/receive events changes states
    - » of the process that sends/receives the message
    - » of the involved channel
  - Internal events
    - » only affects the process at which it occurs

Process Execution (2)
Events

- Notation
  - $\mathbf{e^x_i}$ - the **x***th* event at process $\mathbf{p_i}$
- Sequence of events
  - The **events at a process** are linearly, ordered by their order of occurrence
  - The sequence of events as a result of process $\mathbf{p_i}$ execution: $\mathbf{e_i^1, e_i^2, \ldots e_i^x, e_i^{x+1}, \ldots}$
- Set of events produced by $\mathbf{p_i}$

  $\mathbf{h_i = \{e_i^1, e_i^2, \ldots e_i^x, e_i^{x+1}, \ldots\}}$
- Order relationship among $\mathbf{p_i}$ events is $\mathbf{\rightarrow_i}$
  - expresses causal dependencies among events
- χ_i - process $\mathbf{p_i}$ execution

  $\mathbf{\chi_i = (\ h_i\ , \rightarrow_i\ )}$

Process Execution (3)
Messages

- Notation
  - **send(**m**)** and **rec(**m**)**
    - » send and receive events for a message **m**
- **send** and **receive** primitives
  - Represent information flow between processors
  - Establish *causal dependency* from the *sender process* to the *receiver process*

  $\mathbf{\rightarrow_{msg}}$
  - expression of causal dependency between a pair of corresponding (send, receive) events

  $\mathbf{send(m) \rightarrow_{msg} receive(m)}$

Time–space diagrams

- Evolution of a distributed execution
- Concepts
  - Horizontal line - the progress of a process
  - Dot - indicates an event
  - Slant arrow - indicates a message transfer
- Notes
  1. The execution of an event takes a finite amount of time
  2. Assumption: event execution is atomic (i.e. indivisible and instantaneous) => it is represented as a dot on a process line

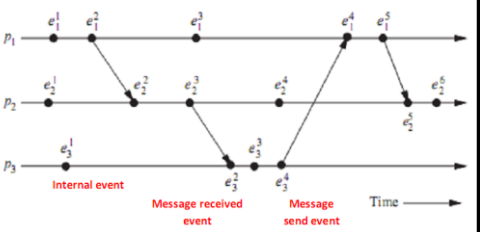


Figure 1 (source [3])

Distributed Application Execution (1)

- Execution of a distributed application
  - Generates a set of distributed events produced by the processes
- Notation
  - **H** - the set of events executed in a distributed computation

$$\mathbf{H} = \mathbf{U}_i \, \mathbf{h}_i$$

Distributed Application Execution (2)

- Causal precedence relationship $\rightarrow$
  - happens-before Lamport relation

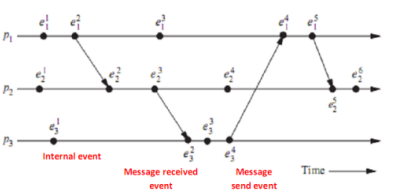
$$\forall e_i^x,\ \forall e_j^y \in H,\quad e_i^x \rightarrow e_j^y \quad \Leftrightarrow \quad \begin{cases} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i=j) \wedge (x<y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \ \wedge \ e_k^z \rightarrow e_j^y \end{cases}$$

- Relationship $\rightarrow$ induces an ireflexive partial order on the events of a distributed computation that is denoted as:

$$\mathcal{H}=(H, \rightarrow)$$

Distributed Application Execution (3)

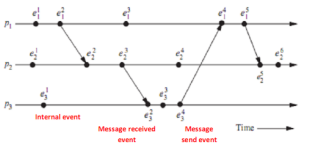
- Meaning of relation $\rightarrow$
  - flow of information (knowledge) in a distributed computation
  - $e_i \rightarrow e_j$ means: all information available at e_i is potentially accessible at e_j
  - Example
    - » e_2^6 in the time-space diagram
    - » e_2^6 event has the knowledge of all other events shown in the figure



Dependent events

- Consider two events $\mathbf{e_i}$ and $\mathbf{e_j}$:
  - **if** ($e_i \rightarrow e_j$)
  - **then**
    - » event $\mathbf{e_j}$ is directly dependent or
    - » event $\mathbf{e_j}$ transitively dependent on event $\mathbf{e_i}$;
- Graphically
  - There is a path along increasing time in space–time diagram that starts at $\mathbf{e_i}$ and ends at $\mathbf{e_j}$
  - The path consists of message arrows and process-line segments
- For example, in the previous time-space diagram:

$e_1^1 \rightarrow e_3^3$ and $e_3^3 \rightarrow e_2^6$



Independent events

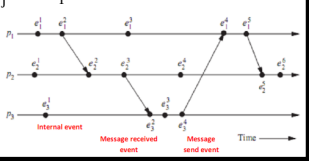
- Consider two events $\mathbf{e_i}$ and $\mathbf{e_j}$;
  - If $\mathbf{e_j}$ does not directly or transitively dependent on $\mathbf{e_i}$
  - Then $\mathbf{e_i}$ and $\mathbf{e_j}$ are independent events
  - Notation: $\mathbf{e_i} \not\rightarrow \mathbf{e_j}$
    - » Event $\mathbf{e_i}$ does not causally affect event $\mathbf{e_j}$ or,
    - » Event $\mathbf{e_j}$ is not aware of the execution of $\mathbf{e_i}$ or any event executed after $\mathbf{e_i}$ on the same process.
- Example (see figure 1):

$e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$

- Rules relating independent events
  - For any two events $\mathbf{e_i}$ and $\mathbf{e_j}$

$e_i \not\rightarrow e_j$ => / $e_j \not\rightarrow e_i$ and

$e_i \rightarrow e_j$ => $e_j \not\rightarrow e_i$



Concurrent events

- For any two events $\mathbf{e_i}$ and $\mathbf{e_j}$
  - **if** $(e_i \not\rightarrow e_j)$ AND $(e_j \not\rightarrow e_i)$,
  - **then** events e_i and e_j are concurrent
- Concurrent relation

  $e_i \parallel e_j$
- Example (see figure 3.1)

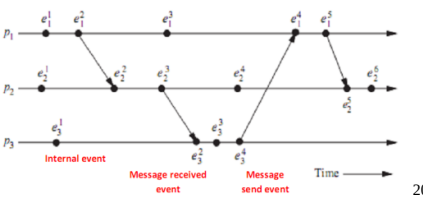
  $e_1^3 \parallel e_3^3$ $\quad$ $e_2^4 \parallel e_3^1$
- Relation $\parallel$ is not transitive;
- $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \Rightarrow \not\; e_i \parallel e_k$
- Example (see figure 1)

  $e_3^3 \parallel e_2^4$ and $e_2^4 \parallel e_1^5$ but $e_3^3 \not\parallel e_1^5$
- For any two events $\mathbf{e_i}$ and $\mathbf{e_j}$ of a distributed execution:

  $\mathbf{e_i \rightarrow e_j}$ **or** $\mathbf{e_j \rightarrow e_i}$ **or** $\mathbf{e_i \parallel e_j}$

Logical and Physical concurrency in a Distributed Computation (1)

- Physical concurrency
  - Events occur at the same instant in physical time
- Logical concurrency
  - Two events are logically concurrent if and only if they do not causally affect each other
  - Two (or more events) may be logically concurrent even though they do not occur at the same physical time
- Example (see figure 1)
  - $\{e_1^3, e_2^4, e_3^3\}$ are logical concurrent
  - They occur at different instants in physical time



Logical and physical concurrency in a Distributed Computation (2)

- Stable computation outcome
  - Assume a set of logically concurrent events
  - The order they occur in physical time (even all at the same time) doesn't influence the computation result
  - $\Rightarrow$ for practical and theoretical purposes, a set of logically concurrent events (that doesn't occur at the same physical time) may be considered as happening at the same physical time

Communication Network Models (1)

- FIFO model
  - Each channel models a first-in first-out message queue
  - message ordering is preserved by a channel
- Non-FIFO model
  - Each channel models an unordered set
  - Sender process adds messages to the set
  - Receiver process removes messages from the set
- Causal ordering (CO)
  - Based on Lamport's "happens before" relation.
  - Property of causal order systems:
  - For any two messages $\mathbf{m_{ij}}$ and $\mathbf{m_{kj}}$

    if (send($\mathbf{m_{ij}}$) $\rightarrow$ send($\mathbf{m_{kj}}$))

    then rec($\mathbf{m_{ij}}$) $\rightarrow$ rec($\mathbf{m_{kj}}$)

Communication Network Models (2)

- CO property
  - Causally Ordered messages to the same destination are delivered in the same order as their causality relation
  - Causally Ordered delivery of messages => FIFO message delivery
- Note:

  $CO \subset FIFO \subset Non\text{-}FIFO$

Communication Network Models (3)

- CO – useful for and simplifies the design of distributed algorithms
  - provides a built-in synchronization
- Example
- Replicated database system
  - For consistency purposes - every process updating a replica receives the updates in the same order to maintain database consistency
  - If CO is not used - each update must be checked to ensure that database consistency is not being violated

Technical University of Cluj-Napoca
Computer Science Department

Distributed Systems

- Master in CS -

Review

- C2 -

Global States and Snapshots

Fall 2017

Global States and Snapshots
Contents

Introduction (1)

- Applications requiring periodically recording of DS state for its analyzing during execution or post mortem
  - transparent failure recovery,
  - distributed debugging,
  - monitoring distributed events,
  - setting distributed breakpoints,
  - protocol verification, etc.

Introduction

- Global State of a DS
  - Collection of the local states of
    - » the processes and
    - » the channels
- Processor State
  - Contents of
    - » processor registers,
    - » stacks,
    - » local memory, threads, status of timeout timer, etc.
- Channel state
  - The set of messages in transit in the channel
- Occurrence of events (internal, send or received):
  - Changes the states of processes and channels
  - Generate transitions in global system state

Process State

- Notation $\mathbf{LS_i^x}$ (local state of process p_i)
  - after the occurrence of event $\mathbf{e_i^x}$ and
  - before the event $\mathbf{e_i^{x+1}}$.
  - The state is the result of $\mathbf{p_i}$ execution of all the events until $\mathbf{e_i^x}$
  - $\mathbf{LS_i^0}$ - the initial state of process $\mathbf{p_i}$

- Notations:

  send(m) $\leq$ $\mathbf{Ls_i^x}$ meaning

  $$\exists y: 1 \leq y \leq x :: e_i^y = send(m)$$

  rec (m) /$\leq$ $\mathbf{Ls_i^x}$ meaning

  $$\forall y: 1 \leq y \leq x :: e_i^y \neq rec(m)$$

  - Read it as: Up to the occurrence of the x event, the i-th processor state has not recorded the **rec(m)** event

Channel State

- Notation $\mathbf{SC_{ij}^{x,y}}$
  - The state of the channel $\mathbf{C_{ij}}$ connecting $\mathbf{p_i}$ to $\mathbf{p_j}$
  - The state of a channel depends on the local states of the processes that are linked by the channel

$$SC_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq LS_i^x \wedge rec(m_{ij}) \not\leq LS_j^y\}$$

- Read it as:

  Channel state $\mathbf{SC_{ij}^{x,y}}$ represents all messages sent by $\mathbf{p_i}$ up to event $\mathbf{e_i^x}$ and which process $\mathbf{p_j}$ had not received until event $\mathbf{e_j^y}$

Messages in transit on $\mathbf{C_{ij}}$

transit(LS_i , LS_j) =

{ m_{ij} | **send**(m_{ij}) $\in$ LS_i AND **rec**(m_{ij}) /$\in$ LS_j }

Global State (GS)
Meaningful GS

- GS – collection of local states of processors and channels

$$GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$$

- A Global Snapshot is **meaningful** when
  - the states of all the components of the DS are recorded at the same time instant
- This is possible when
  - the local clocks at processes were perfectly synchronized or
  - there was a global system clock that could be instantaneously read by the processes.
- Both conditions are impossible to be fulfilled

Global State
Consistent GS (1)

- The state of DS components may be recorded at different time instants => set of states
- **Consistent Global States**
  - If every message that is recorded in the set of states as received is also recorded as sent
- **Transit-less global state**
  - All channels are recorded as empty in such a global state
- **Strongly consistent global state**
  - Consistent + Transit-less

Global State
Consistent GS (2)

- GS is **consistent global state** if its satisfies conditions C1 and C2
- **C1** (message conservation law):
  send(m_{ij}) $\in$ LS_i =>
  $m_{ij} \in SC_{ij}$ XOR rec(m_{ij}) $\in$ LS_j

  Every message recorded as sent by $\in$ LS_i should be in the channel or already received by LS_j
- **C2** (cause-effect law):
  send(m_{ij}) /$\in$ LS_i =>
  m_{ij} /$\in$ SC_{ij} AND rec(m_{ij}) /$\in$ LS_j
  - If the message was not sent it cannot be in the channel or already received
  - A consistent global state captures the notion of causality a message cannot be received if it was not sent

Example

- Example of Distributed Execution
- GS1 = {LS_1^1 , LS_2^3 , LS_3^3 , LS_4^2 }
  - Inconsistent - state of p2 has recorded the receipt of message m12, however, the state of p1 has not recorded it's send.
- GS2 = {LS_1^2 , LS_2^4 , LS_3^4 , LS_4^2 }
  - Consistent - All the channels are empty except C21 that contains the message m21
- GS3 = {LS_1^2, LS_3^2, LS_4^3, LS_2^4 }
  - Strongly consistent

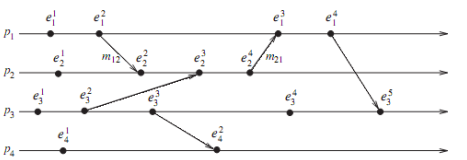


Figure 2 (source [3])

Cuts in Distributed Computations

- Cut
  - A zig-zag line in the time-space diagram
  - Joins one arbitrary point on each process of a distributed computation
  - A cut **C** slices the space-time diagram and the set of events into PAST and FUTURE
- PAST(C)
  - All the events at the left of the cut C
- FUTURE(C)
  - All the events at the right of the cut C

Cuts and Global States

- Cut and GS relationship
  - Every cut corresponds to a GS
  - Every GS can be graphically represented as a cut in the time-space diagram
- Consider $e_i^{Max\_PAST_i(C)}$ as the latest event at process p_i that is in the PAST of a cut C
- The GS represented by the Cut C is

  $\{ \mathbf{U}_i LS_i^{Max\_PAST(C)}, \mathbf{U}_{j,k} SC_{jk}^{yj,zk} \}$

  where

  $SC_{jk}^{yj,zk} = \{ m \mid send(m) \in PAST(C)$
  AND $rec(m) \in FUTURE(C) \}$

Consistent and inconsistent cuts (1)

- Consistent cut C
  - Corresponds to a consistent GS
    - » Every message received in the PAST of the cut was sent in the PAST of that cut
    - » All messages crossing the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state
- Inconsistent cut
  - A cut is inconsistent if a message crosses the cut from the FUTURE to the PAST

Consistent and inconsistent cuts (2)
Example

- C1 (corresponds to GS1) - inconsistent cut
  - **m1** goes from the future to the past
- C2 (corresponds to GS2) - consistent cut
  - **m4** must be stored in the state of $\mathbf{C_{12}}$
- Cuts in a space–time diagram
  - A powerful graphical aid in representing and reasoning about global states of a computation
- In a consistent snapshot, all the recorded local states of processors are concurrent
  - The recorded local state of no process causally affects the recorded local state of any other processes

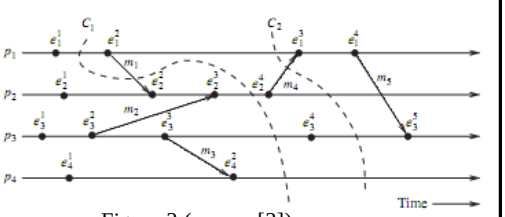


Figure 3 (source [3])

Problems in recording global states (1)

- Assume the **availability of a global physical clock**
- All messages are time stamped with the sender's clock
- Procedure for recording a consistent global snapshot of a distributed system
  - The initiator of the snapshot collection decides a future time at which the snapshot is to be taken
  - Broadcasts this time to every process
  - All processes take their local snapshots at that instant in the global time
  - The snapshot of channel $\mathbf{C_{ij}}$
    - » all the messages that process $\mathbf{p_j}$ receives after taking the snapshot and whose timestamp is smaller than the time of the snapshot. (All messages are time stamped with the sender's clock)
    - » If channels are not FIFO, a termination detection scheme will be needed to determine when to stop waiting for messages on channels
- In real DS a global physical clock is not available

Problems in recording global states (2)

- Recording of a consistent global snapshot of a real DS
- **P1:** How to distinguish between the messages to be recorded in the snapshot (either in a channel state or a process state) from those not to be recorded ?
  - Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (**C1**)
  - Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (**C2**)
- **P2:** How to determine the instant when a process takes its snapshot?
  - A process $\mathbf{p_j}$ must record its snapshot before processing a message $\mathbf{m_{ij}}$ that was sent by the process $\mathbf{p_i}$ after recording its snapshot

Snapshot algorithms
Introduction

- Objective
  - Present the representative snapshot algorithms for distributed systems
- Consider a DC and a snapshot algorithm
- Two types of messages should be considered
  - computation messages
    - » exchanged by the underlying DS application
  - control messages
    - » exchanged by the snapshot algorithm
- Execution of a snapshot algorithm is transparent to the underlying application

Snapshot algorithms for FIFO channels
Chandy-Lamport algorithm

- *Chandy-Lamport* algorithm (1985) uses a control message, called **marker** (the paper presenting this algorithm is posted in the course directory)
- After a site has recorded its snapshot, it sends a marker over all of its outgoing channels before sending out any more messages
- Due to FIFO channels
  - The marker separates the messages in the channel into (this addresses the issue **P1**)
    - » Messages to be included in the snapshot (i.e., channel state or process state) from
    - » Messages that should not be recorded in the snapshot. This addresses issue P1

Snapshot algorithms for FIFO channels
Chandy-Lamport algorithm

- Role of markers
  - Delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the **condition C2**
  - Since all messages that follow a marker on channel $\mathbf{C_{ij}}$ have been sent by process $\mathbf{p_i}$ after $\mathbf{p_i}$ has taken its snapshot,

    => process $\mathbf{p_j}$ must record its snapshot no later than when it receives a marker on channel $\mathbf{C_{ij}}$
  - In general, a process must record its snapshot no later than when it receives a marker on any of its incoming channels. This addresses **issue P2**

Snapshot algorithms for FIFO channels
Chandy-Lamport algorithm

Marker sending rule for process $\mathbf{p_i}$

(1) Process p_i records its state.

(2) For each outgoing channel C on which a marker has not been sent, p_i sends a marker along C before p_i sends further messages along C

Marker receiving rule for process pj

On receiving a marker along channel C:

if $\mathbf{p_j}$ has not recorded its state

then

- Record the state of C as the empty set
- Execute the "marker sending rule"

else

- Record the state of C as the set of messages received along C after $\mathbf{p_j}$'s state was recorded and before $\mathbf{p_j}$ received the marker along C

Snapshot algorithms for FIFO channels
Chandy-Lamport algorithm

- **Algorithm discussion**
- The algorithm can be initiated by any process by executing the marker sending rule
- The algorithm terminates after each process has received a marker on all of its incoming channels
- Assembling the global state
  - The recorded local snapshots can be assembled to create the global snapshot in several ways
  - Policy 1
    - » Each process send its local snapshot to the initiator of the algorithm
  - Policy 2
    - » Each process send the information it records along all outgoing channels, and
    - » Each process receiving such information for the first time propagate it along its outgoing channels
    - » All the local snapshots get disseminated to all other processes and all the processes can determine the global state

Snapshot algorithms for FIFO channels
Chandy-Lamport algorithm

- **Algorithm discussion (cont.)**
- Multiple processes can initiate the algorithm concurrently
- If multiple processes initiate the algorithm concurrently
  - Each initiation needs to be distinguished by using unique markers
  - Different initiations by a process are identified by a sequence number

Snapshot algorithms for FIFO channels

Variations of the Chandy-Lamport algorithm

- Different other proposals refine and optimize the basic algorithm.
- Examples
  - **Spezialetti and Kearns** algorithm
    - » optimizes concurrent initiation of snapshot collection and
    - » efficiently distributes the recorded snapshot
  - **Venkatesan**'s algorithm (the paper is posted in the class directory)
    - » optimizes the basic snapshot algorithm to efficiently record repeated snapshots of a distributed system that are required in recovery algorithms with synchronous check pointing

Snapshot algorithms for **non FIFO** channels

- A FIFO system
  - ensures that all messages sent after a marker on a channel will be delivered after the marker

  => condition C2 is satisfied in the recorded snapshot if LS_i, LS_j , and SC_{ij} are recorded as described in the Chandy–Lamport algorithm
- In a non-FIFO system, the problem of global snapshot recording is more complex
  - A marker cannot be used to discriminate messages into those to be recorded in the global state from those not to be recorded in the global state.
  - In such systems, different techniques must be used to ensure that a recorded global state satisfies condition C2

Snapshot algorithms for **non FIFO** channels

- In a non-FIFO system, to record a consistent GS is necessary
  - some degree of inhibition (i.e., temporarily delaying the execution of an application process or delaying the send of a computation message) or
  - piggybacking of control information on computation messages to capture out-of-sequence messages
- **Helary** algorithm (non-FIFO algorithm)
  - uses message inhibition
  - Paper posted in the class directory
- **Lai and Yang, Li et al**. and **Mattern** non-FIFO algorithms
  - use message piggybacking to distinguish computation messages sent after the marker from those sent before the marker
  - Papers posted in course directory

Snapshot algorithms for CO delivery systems

- Global snapshot recording algorithms assume that the underlying system supports Causal Order message delivery
- The Causal Order message delivery property provides a built-in message synchronization
  => Snapshot algorithms for such systems are considerably simplified
- Representative algorithms
  - Acharya–Badrinath
  - Alagar–Venkatesan
- These algorithms do not send control messages (i.e., markers) on every channel and are simpler than the snapshot algorithms for a FIFO system
- Several protocols for implementing causal ordering have been proposed by Birman, Raynal and Schiper

Monitoring global state

- System state is given by the values of a set of variables distributed on DS processes
  - System state can be expressed as a predicate on variables distributed across processes
- The need to evaluate system state
  - Example: for debugging a distributed program
- The monitoring approach
  - Instead of recording and evaluating snapshots at regular intervals
  - More efficient to monitor changes to the variables that affect the predicate and
  - Evaluate the predicate only when some component variable changes
- Simultaneous regions technique
  - Proposed by Spezialetti and Kearns
  - Objective: consistent monitoring of distributed systems to detect global predicates
  - Basic idea: a process whose local variable is component of a global predicate informs monitor whenever the variable value changes

Check pointing (1)

- Many applications (seen before) require that local process states are periodically recorded and analyzed during execution or post mortem
- Local process checkpoint
  - A saved intermediate state of a process during its execution
- A global snapshot of a DS
  - A set of local checkpoints (one taken from each process)
  - Represents a snapshot of the DC execution at some instant

Check pointing (2)

- A global snapshot is consistent
  - if there is no causal path between any two distinct checkpoints in the global snapshot

  => A consistent snapshot consists of a set of local states that occurred concurrently or had a potential to occur simultaneously
  - This condition for the consistency of a global snapshot (that no causal path between any two checkpoints) is only the necessary condition but it is not the sufficient condition

Check pointing (3)

- **Zig-zag** path developed by **Netzer** and **Yu**
  - define a generalization of the Lamport's happens before relation, called a **zigzag path** for describing the necessary and sufficient conditions for constructing a consistent snapshot from a set of checkpoints S
- **Manivannan–Netzer–Singhal**
  - developed an algorithm that explicitly computes all consistent snapshots that include a given set a set of checkpoints S

Technical University of Cluj-Napoca
Computer Science Department

Distributed Systems

- Master in CS -

Review

C3

Time and Causality in Distributed Systems

Fall 2017

Time and Causality in DS
Contents

- Causality
- Systems of Logical Time (overview)
- A System of Logical Clocks
- Scalar Time (Lamport)
- Vector Time

Causality (1)

- Causality in day by day life
  - Used in scheduling, planning and actions
  - Causality is tracked using physical time
  - Loosely synchronized clocks
    - » The occurrence of events is low
- In DS the occurrence of events is high
  - The causality of events suffers and could not be accurately captured
- Event causality – fundamental concept in design and analysis of parallel and distributed computing and operating systems
- In DS - is not possible to have global physical time (only an approximation of it)

Causality (2)

- Causality in distributed computations
  - Logical clocks
  - A system of logical clocks should:
    - » Capture the causality relation between events produced by a program execution
    - » Capture the fundamental monotonicity property of causal relation
- System of logical clocks
  - Every process has a logical clock that is advanced using a set of rules
  - Every event is assigned a timestamp
    - » The causality relation between events can be generally inferred from their timestamps
  - The assigned timestamps obey the fundamental monotonicity property, i.e.:
    - » **if** an event **a** causally affects an event **b**, **then** the timestamp of **a** is smaller than that of **b**

Causality (3)

- Causality in distributed algorithms design
  - In mutual exclusion algorithms to ensure liveness and fairness
  - In maintaining consistency in replicated databases
  - In designing correct deadlock detection algorithms
- Causality in tracking of dependent events
  - In distributed debugging
    - » Knowledge of the causal dependency among events helps construct a consistent state for resuming re-execution;
  - In failure recovery
    - » It helps build a checkpoint;
  - In replicated databases
    - » It aids in the detection of file inconsistencies in case of a network partitioning

Causality (4)

- Causality in knowledge about the progress
  - Knowledge of the causal dependency among events helps measure the progress of processes in the distributed computation
  - Useful in discarding obsolete information, garbage collection, and termination detection
- Causality in concurrency measure
  - Knowledge of how many events are causally dependent is useful in measuring the amount of concurrency in a computation
  - All events that are not causally related can be executed concurrently

  => Analysis of the causality in a computation gives an idea of the concurrency in the program

Systems of Logical Time (overview)

- Ways to implement logical time
- The following techniques have been proposed to capture causality between events of a distributed computation
  - Scalar time (Lamport's scalar clocks)
    - » Time is represented by non-negative integers
  - Vector time
    - » Time is represented by a vector of non-negative integers
  - Matrix time
    - » Time is represented as a matrix of non-negative integers
  - Virtual time
    - » Time-warp mechanism

A system of Logical Clocks
Definition

- The elements of a system of logical clocks
  - **T** - a time domain and
  - **C** - a logical clock function
- Relationships between the system of logical clock elements

  (T, <) partially ordered set over the < relation
  - » Elements of **T** are partially ordered by the < relationship
  - » Relation < is usually called the **happened before** or **causal precedence**.
  - » **Note.** Relation < is analogous to the **earlier than** relation provided by the physical time

  C: H → T
  - » The logical clock function **C** maps an event **e** in H (a distributed system execution) to an element in the time domain T, denoted as **C(e)**
  - » **C(e)** - the **timestamp** of **e**
  - **C** should satisfy the monotonicity property (called the clock **consistency** condition):

    $$e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$$
  - **Strong consistent** systems of clocks condition:

    $$e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$$

A system of logical clocks
Implementation

- Two problems should be addressed when implementing logical clocks
  - Data structures for representing logical time
    - » Should be implemented at the level of each local process
  - The data structure updating rules for ensuring the consistency condition

A system of logical clocks
Implementation – Data Structure

- Data structure (at the level of each process p_i) should allow to represent
  - A local logical clock $\mathbf{lc_i}$
    - » helps process $\mathbf{p_i}$ measure its own progress
  - A global logical clock, $\mathbf{gc_i}$
    - » A representation of process $\mathbf{p_i}$'s local view of the logical global time.
    - » It allows the local process to assign consistent timestamps to its local events.
  - $\mathbf{lc_i}$ is a part of $\mathbf{gc_i}$

A system of logical clocks
Implementation – The Rules

- Objective
  - Ensures a consistency management of a process's logical clock $\mathbf{lc_i}$
    - » as a result also ensures a consistent view of the global time

Rule R1

- How a process local logical clock is updated when an event is executed (internal event, send event, receive event)

Rule R2

- How a process updates the global logical clock and global progress
  - » Shows what information about the logical time is piggybacked in a message and
  - » how this information is used by the receiving process to update its view of the global time

A system of logical clocks
Implementation - Conclusion

- Each particular implementation of a system of logical clocks should specify how
  - The data structure is represented
  - The **R1** and **R2** rules are represented and implemented
- Each particular logical clock system provides its users with some additional properties

Scalar time

Definition – Data structure

- Proposed by Lamport (1978)
  - Attempt to totally order events in a distributed system
- Time domain representation
  - Set of non-negative integers
  - For a process $\mathbf{p_i}$ an integer variable $\mathbf{C_i}$ represents both
    - » the logical local clock of the process
    - » its local view of the global time

Scalar time

Definition – The rules (1)

- Clock updating rules
- **Rule R1**
  - Before executing an event (send, receive, or internal), process $\mathbf{p_i}$ executes the following:

$$\mathbf{C_i := C_i + d} \quad (d > 0)$$

    - » **d** can have a different values, every time **R1** is executed
    - » **d** value may be application-dependent.
    - » Usually **d = 1** (the lowest rate that allows to uniquely identify the time of each at a process)

Scalar time

Definition – The rules (2)

- **Rule R2**
  - Each message piggybacks the clock value of its sender at sending time
- When process $\mathbf{p_i}$ receives a message with timestamp $\mathbf{C_{msg}}$, it executes the following actions:
  1. $C_i := \max(C_i, C_{msg})$;
  2. execute R1;
  3. deliver/processes the message

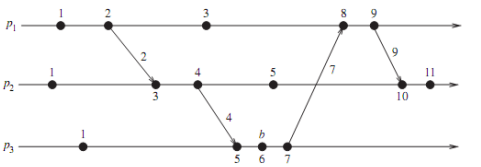


Fig. 4 - Evolution of scalar time (source [3])

Scalar time

Basic properties – consistency property

- Scalar clocks satisfy the monotonicity

  => the consistency property is also satisfied, i.e.:

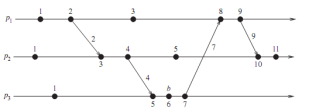
  – for two events e_i and e_j,

  $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$

Scalar time

Basic properties – total ordering (1)

- Scalar clocks - used to totally order events in a distributed system
- Totally ordered events main problem
  - Two or more events at different processes may have an identical timestamp
  - In Fig the third event of process P_1 and the second event of process P_2 have identical scalar timestamp.
    - » A tie-breaking mechanism is needed to order such events
  - A tie is broken as follows:
    - » A tie among events with identical scalar timestamp is broken on the basis of their process identifiers
    - » The lower the process identifier in the ranking, the higher the priority
    - » **Note.** Process identifiers are ordered
  - The **timestamp of an event** in the DS is denoted by a tuple **(t, i)** where
    - » **t** - time of occurrence and
    - » **i** - the process where it occurred



Scalar time

Basic properties – total ordering (2)

- Total order relation $\prec$ on two events
  - Event **x** with timestamp (h, i)
  - Event **y** with timestamp (k, j)

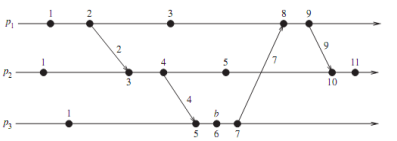
  $x \prec y \Leftrightarrow (h < k$ **or** $(h = k$ **and** $i < j))$
- Since events that occur at the same logical scalar time are independent (not causally related) => they can be ordered using any arbitrary criterion without violating the causality relation $\rightarrow$
  - Therefore, a total order is consistent with the causality relation "$\rightarrow$"
- Note that $x \prec y \Rightarrow x \rightarrow y \ \vee \ x \parallel y$
- A total order is generally used to ensure liveness properties in distributed algorithms
  - The requests are time stamped and served according to the total order based on these timestamps

Scalar time

Basic properties – Event counting

- Height of event property of scalar time
  - If event **e** has a timestamp **h**

  => **h−1** represents the **minimum logical duration**, counted in units of events (i.e. units of **d)** , required before producing the event **e**
- In other words:
  - Assuming d =1 => **h-1** events have been produced sequentially before the event **e** regardless of the processes that produced these events.
- Example
  - in Fig, five events precede event **b** on the longest causal path ending at **b**

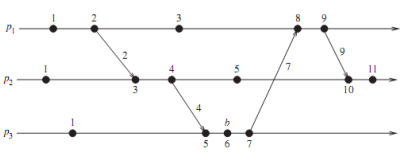


Scalar time

Basic properties – No strong consistency (1)

- The system of scalar clocks is not strongly consistent, i.e.:
  - for two events $\mathbf{e_i}$ and $\mathbf{e_j}$,

    $C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$
- Example (Fig. 1) - The third event of process P_1 has smaller scalar timestamp than the third event of process P_2
  - However, the former did not happen before the latter



Scalar time

Basic properties – No strong consistency (2)

- Why scalar clocks are not strongly consistent ?
  - Because both the logical local clock and logical global clock of a process are represented into one value

  => the loss causal dependency information among events at different processes
  - Example (Fig. 4)
    - » When process P2 receives the first message from process P1, it updates its clock to 3, forgetting that the timestamp of the latest event at P1 on which it depends is 2.

Vector time

Definition – Data structure

- Research of Fidge, Mattern, and Schmuck
- Time domain representation
  - A set of **n-dimensional** non-negative integer vectors
- For each process $\mathbf{p_i}$ is defined vector

  $\mathbf{vt_i\ [1..\ n]}$
  - $\mathbf{vt_i\ [i]}$ is the local logical clock of $\mathbf{p_i}$ and describes the logical time progress at process $\mathbf{p_i}$
  - $\mathbf{vt_i\ [j]}$ represents process $\mathbf{p_i}$'s latest knowledge of process $\mathbf{p_j}$ local time
    - » **if** $vt_i\ [j] = x$,

      then process p_i knows that local time at process p_j has progressed till x
- The vector $\mathbf{vt_i}$
  - represents $\mathbf{p_i}$'s view of the global logical time
  - is used to timestamp events

Vector time
Definition – The rules

Rule R1

Before executing an event, $\mathbf{p_i}$ updates its local logical time as follows:

$$vt_i[i] := vt_i[i] + d \qquad (d > 0).$$

Rule R2

- Each message **m** is piggybacked with the vector clock **vt** of the sender process at sending time
- On receiving the message **(m,vt)**, process $\mathbf{p_i}$ executes the following sequence of actions:

A1. update its global logical time as follows:

$$1 \leq k \leq n \ : \ vt_i[k] := max(vt_i[k], vt[k]);$$

A2. execute R1;
A3. deliver/process the message m.

Vector time
Definition – The rules

- The **timestamp** associated with an event is the value of the vector clock of its process when the event is executed
- Example (Figure 5)
  - Initially, a vector clock is [0, 0, 0, … 0]
  - See vector clocks progress with the increment d = 1
- Comparing two vector timestamps, vh and vk:
  - The following relations are used

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$

$$vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$$

$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$

$$vh \parallel vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh).$$

Vector time

Definition – The rules

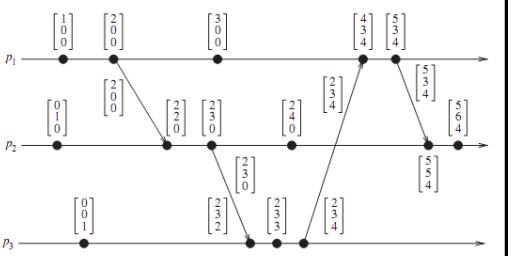


Fig. 5 - Evolution of vector time (source [3])

Vector time

Basic properties – Strong consistency

- The system of vector clocks is strongly consistent
  => by examining the vector timestamp of two events, we can determine if the events are causally related
- The necessary precondition
  – n - the total number of processes in the distributed computation
  – The dimension of vector clocks cannot be less than n (Charron–Bost research)

Vector time
Basic properties – Applications

- Vector time tracks causal dependencies => a wide variety of applications
- Examples of applications
  - distributed debugging,
  - implementations of causal ordering communication
  - causal distributed shared memory
  - establishment of global breakpoints
  - determining the consistency of checkpoints in optimistic recovery

Vector time
Efficient implementation of vector clocks

- Large # of processes in a DC
  - Vector clocks requires piggybacking of large amount of information in messages for disseminating time progress and updating clocks
  - The message overhead grows linearly with the number of processors in the system
    - » When there are thousands of processors in the system => message size becomes huge even if there are only a few events occurring in few processors

    => necessity of efficient techniques to maintain vector clocks
    - » similar techniques can be used to efficiently implement matrix clocks

Vector time
Efficient implementation of vector clocks

- Vector size
  - If vector clocks have to satisfy the strong consistency property
    => vector timestamps must be at least of size n, the total number of processes [shown by Charron-Bost]
  - In general the size of a vector timestamp is the number of processes involved in a distributed computation
  - Several optimizations are possible to implement vector clocks efficiently (see the references)

Vector time
Efficient implementation of vector clocks

- Singhal–Kshemkalyani's differential technique
- Fowler –Zwaenepoel direct-dependency technique
- Jard–Jourdan's adaptive technique