RAMAIAH
Institute of Technology

M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Distributed Systems
Course Code: CSE20/CSE751
Credits: 3:0:0

Term: Oct 2021-Feb 2022

Faculty:
Sini Anna Alex

Introduction and Uses of Predicate Detection

- Industrial process control, distributed debugging, computer-aided verification, sensor networks
- E.g., ψ defined as $x_i + y_j + z_k < 100$
- Different from global snapshots: global snapshot gives one of the values that could have existed during the execution
- Stable predicate: remains true once it becomes true, i.e., $\phi \implies \Box\phi$
  - predicate ϕ at a cut C is stable if:

$$(C \models \phi) \implies (\forall C' \mid C \subseteq C', C' \models \phi)$$

  - E.g., deadlock, termination of execution are stable properties

Stable Properties

- Deadlock: Given a Wait-For Graph $G = (V, E)$, a *deadlock* is a subgraph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$ and for each i in V', i remains blocked unless it receives a reply from some process(es) in V'.
  - (local condition:) each deadlocked process is locally blocked, and
  - (global condition:) the deadlocked process will not receive a reply from some process(es) in V'.
- Termination of execution: Model active and passive states, and state transitions between them. Then execution is terminated if:
  - (local condition:) each process is in passive state, and
  - (global condition:) there is no message in transit between any pair of processes.
- Repeated global snapshots is not practical!
- Utilize a 2-phased approach of observing potentially inconsistent global states.

Two-phase Observation of Global States

- In each state observation, all local variables used to define the local conditions, as well as the global conditions, are observed.
- Two potentially inconsistent global states are recorded consecutively, such that the second recording is initiated after the first recording has completed. Stable property true if:
  - The variables on which the local conditions as well as the global conditions are defined have not changed in the two observations, as well as between the two observations.
- Recording 2 snapshots serially via ring/tree/flat-tree based algorithms (Chap 8).

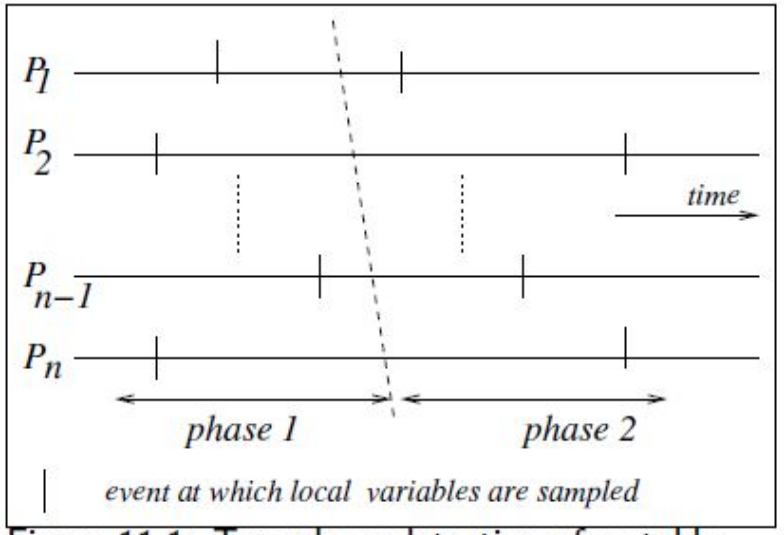


Figure 11.1: Two-phase detection of a stable property.

None of the variables changes between the two observations

$\Rightarrow$ after the termination of the first observation and before the start of the second observation, there is an instant when the variables still have the same value.

$\Rightarrow$ the stable property will necessarily be true.

Unstable Predicates: Challenges in Detection

Challenges:

- unpredictable propagation times, unpredictable process scheduling ⇒
  - multiple executions pass through different global states;
  - predicate may be true in some executions and false in others
- No global time ⇒
  - monitor finds predicate true in a state but predicate may never have been true (at any instant)
  - even a true predicate may be undetected due to intermittent monitoring

Observations:

- examine all states in an execution ⇒ define predicate on observation of entire execution
- Multiple observations of same program may pass thru' different global states; predicate may be true in some observations but not others ⇒ define predicate on all the observations of the distributed program

Modalities on Predicates

- *Possibly*(ϕ): There exists a consistent observation of the execution such that predicate ϕ holds in a global state of the observation.
- *Definitely*(ϕ): For every consistent observation of the execution, there exists a global state of it in which predicate ϕ holds.

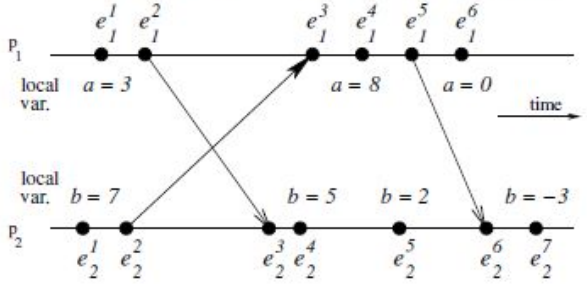
Centralized Algorithm for Relational Predicates (contd.)

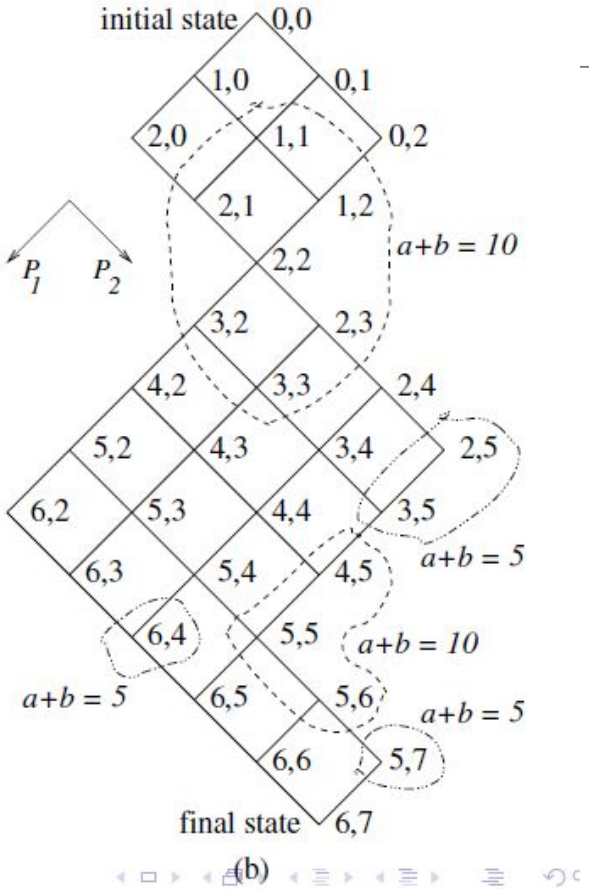
Definitely(ϕ):

- Replacing line (1a) by: "(some state in $Reach_\phi$ satisfies $\neg\phi$)" will not work!
- The algorithm examines the state lattice level-by-level:
  1. Tracks states at each level in which ϕ is not true
  2. The tracked states at a level have to be reachable from states at previous level satisfying (1) and this property (2) recursively.
- $Reach\_Next_\phi$ at level *lvl* contains the set of states at level *lvl* that are reachable from the initial state *without* passing through any state satisfying ϕ.
- return(1) if $Reach\_Next(\phi)$ becomes $\emptyset$, else return(0).
- In example, at $lvl = 11$, $Reach\_Next(\phi)$ becomes empty and *Definitely*(ϕ) is detected.

Conjunctive Predicates

- ϕ can be expressed as the conjunction $\wedge_{i \in N} \phi_i$, where ϕ_i is local to process i.
- If ϕ is false in any cut C, then there is at least one process i such that the local state of i in cut C will never form part of any other cut C' such that ϕ is true in C'.
- If ϕ is false in some cut C, we can advance the local state of at least one process to the next event, and then evaluate the predicate in the resulting cut
- This gives a $O(mn)$ time algorithm, where m is the number of events at any process.
- In example, $Possibly(a = 3 \wedge b = 2)$ and $Definitely(a = 3 \wedge b = 7)$ and true.





Detecting Conjunctive Predicates

- Global state-based approach: $O(mn)$ time
- Interval-based approach: interval X represents duration in which ϕ_i true at i. Standard min and max semantics

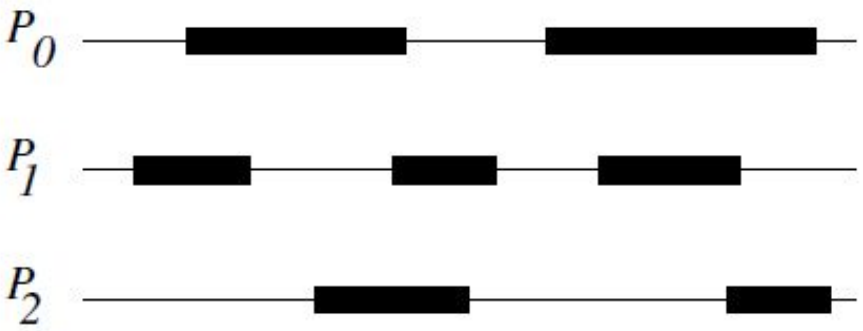


Fig 11.5

Optimization: If no send or receive between start of interval and the end of next interval at that process, the intervals have exact same relation w.r.t. other intervals at other processes.

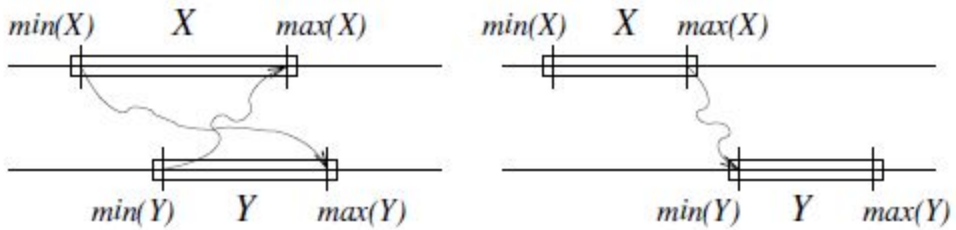
Detecting Conjunctive Predicates, over Multiple Processes

For two processes:

- $Definitely(\phi) : min(X) \prec max(Y) \bigwedge min(Y) \prec max(X)$
- $\overline{Possibly(\phi)} : max(X) \prec min(Y) \bigvee max(Y) \prec min(X)$

For multiple processes:

- $Definitely(\phi) : \bigwedge_{i,j \in N} Definitely(\phi_i \bigwedge \phi_j)$
- $Possibly(\phi) : \bigwedge_{i,j \in N} Possibly(\phi_i \bigwedge \phi_j)$



(a) Definitely(phi) (b) $\overline{Possibly(phi)}$

State-based Algorithm for *Possibly*(ϕ) (conjunctive ϕ)

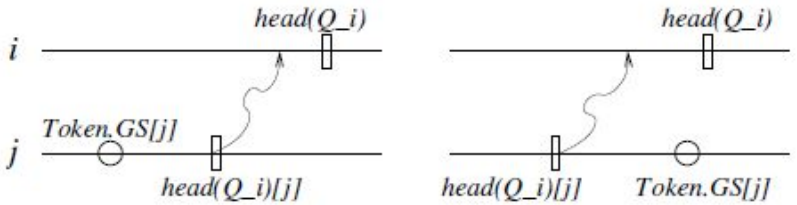
Let m be # local states at any process; let M be # messages sent in the execution

- Termination: when $Valid[j] = 1$ for all j
- Time complexity: $O(n^2 m)$
- Space complexity: $O(n^2 m)$
- Message complexity: $2M$ control messages, each of size n

Distributed state-based algorithm for *Possibly*(ϕ)

(conjunctive ϕ)

- $Token.GS[1..n]$ gives the timestamp of the latest cut under consideration as a candidate solution.
- $Token.Valid[1..n]$. $Token.Valid[i] = 0$ implies all P_i local states up to $Token.GS[i]$ cannot be part of the solution. So from Q_i, consider the earliest local state such that local timestamp is greater than $Token.GS[i]$.
- $Token.GS[i]$, $.Valid[i]$ entries are set accordingly.
- Consistency checks made between $head(Q_i)[j]$ and $Token.GS[j]$ (for all j), to determine whether the various $Token.Valid$ entries should be 1 or 0.
- Token passed to any process for which $Token.Valid = 0$.



P_i tests whether P_j's candidate local state $Token.GS[j]$ is consistent with $head(Q_i)[i]$, which is assigned to $Token.GS[i]$. The two possibilities are illustrated. (a) Not consistent. (b) Consistent.

Stable predicates

Stable predicates

Deadlock

A deadlock represents a system state where a subset of the processes are blocked on one another, waiting for a reply from the other processes in that subset. The waiting relationship is represented by a wait-for graph (WFG) where an edge from i to j indicates that process i is waiting for a reply from process j. Given a wait-for graph $G = (V, E)$, a *deadlock* is a subgraph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$ and for each process i in V', the process i remains blocked unless it receives a reply from some process(es) in V'. There are two conditions that characterize the deadlock state of the execution:

- (local condition:) each deadlocked process is locally blocked, and
- (global condition:) the deadlocked process will not receive a reply from some process(es) in V'.

Termination

Termination of an execution is another stable property, and is best understood by viewing a process as alternating between two states: *active state* and *passive state*. An *active* process spontaneously becomes *passive* when it has no further work to do; a *passive process* can become *active* only when it receives a message from some other process. If such a message arrives, then the process becomes *active* by doing CPU processing and maybe sending messages as a result of the processing. An execution is *terminated* if each process is *passive*, and will not become active unless it receives more messages. There are two conditions that characterize the termination state of the execution:

- (local condition:) each process is in passive state; and
- (global condition:) there is no message in transit between any pair of processes.

Unstable predicates

Unstable predicates

An *unstable* predicate is a predicate that is not stable and hence may hold only intermittently The following are some of the several challenges in detecting unstable predicates:

- Due to unpredictable message propagation times, and unpredictable scheduling of the various processes on the processors under various load conditions, even for deterministic executions, multiple executions of the same distributed program may pass through different global states. Further, the predicate may be true in some executions and false in others.
- Due to the non-availability of instantaneous time in a distributed system:
  - even if a monitor finds the predicate to be true in a global state, it may not have actually held in the execution;
  - even if a predicate is true for a transient period, it may not be detected by intermittent monitoring.

  Hence, periodic monitoring of the execution is not adequate.

These challenges are faced by snapshot-based algorithms as well as by a central monitor that evaluates data collected from the monitored processes. To address these challenges, we can make two important observations [8,18].

- It seems necessary to examine all the states that arise in the execution, so as not to miss the predicate being true. Hence, it seems useful to define predicates, not on individual states, but on the observation of the entire execution.
- For the same distributed program, even given that it is deterministic, multiple observations may pass through different global states. Further, a predicate may be true in some of the program observations but not in others. Hence it is more useful to define the predicates on all the observations of the distributed program and not just on a single observation of it.

Conjunctive predicates

The predicates considered so far are termed *relational predicates* because the predicate can be an arbitrary relation on the variables in the system. A predicate ϕ is a *conjunctive predicate* if and only if ϕ can be expressed as the conjunction $\bigwedge_{i \in N} \phi_i$, where ϕ_i is a predicate local to process i. For a wide range of applications, the predicate of interest can be modeled as a conjunctive predicate. Conjunctive predicates have the following property:

- If ϕ is false in any cut C, then there is at least one process i such that the local state of i in cut C will never form part of any other cut C' such that ϕ is true in C'. More formally, this property of a conjunctive predicate ϕ is defined as the following:

$$C \not\models \phi \Longrightarrow \exists i \in N, \forall C' \in Cuts, C' \not\models \phi, \text{ where } C'[i] = C[i].$$

Interval-based centralized algorithm for conjunctive predicates

Conjunctive predicates are a popular class of predicates. Conjunctive predicates have the advantage that each process can locally determine whether the local component ϕ_i is satisfied; if not, the local state cannot be part of any global state satisfying ϕ. This has the following implication: starting with the initial state, we examine global states. If ϕ is not satisfied, then the local state of at least one process can be advanced and the next global state is examined. Either ϕ is satisfied, or we repeat the step. Within mn steps, we will have examined all necessary global states, giving a $O(mn)$ upper bound on the time complexity.

$$Definitely(\phi) \text{ if and only if } \bigwedge_{i,j \in N} Definitely(\phi_i \wedge \phi_j),$$

$$Possibly(\phi) \text{ if and only if } \bigwedge_{i,j \in N} Possibly(\phi_i \wedge \phi_j).$$

Consensus and agreement algorithms

Failure models Among the n processes in the system, at most of processes can be faulty. A faulty process can behave in any manner allowed by the failure model assumed. The various failure models – fail-stop, send omission and receive omission, and Byzantine failures.

Network connectivity The system has full logical connectivity, i.e., each process can communicate with any other by direct message passing.

• **Sender identification** A process that receives a message always knows the identity of the sender process.

Channel reliability The channels are reliable, and only the processes may fail (under one of various failure models).

Agreement variable The agreement variable may be boolean or multivalued, and need not be an integer.

The Byzantine agreement problem

- **Agreement** All non-faulty processes must agree on the same value.
- **Validity** If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source.
- **Termination** Each non-faulty process must eventually decide on a value.

The consensus problem

- **Agreement** All non-faulty processes must agree on the same (single) value.
- **Validity** If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.
- **Termination** Each non-faulty process must eventually decide on a value.

Agreement in a failure-free system (synchronous or asynchronous)

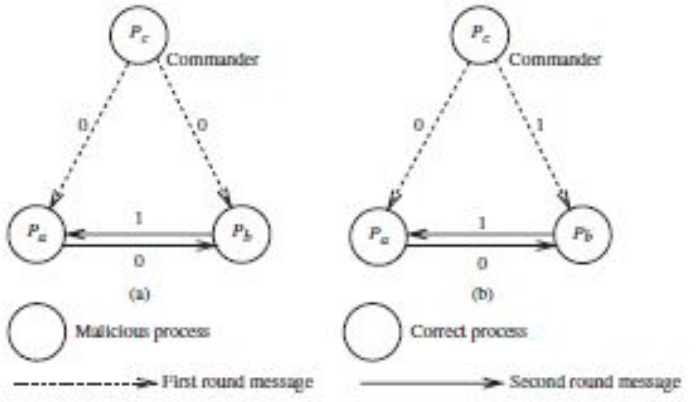
```
(global constants)
integer: f;          // maximum number of crash failures tolerated
(local variables)
integer: x ← local value;

(1)   Process P_i (1 ≤ i ≤ n) executes the consensus algorithm for up to
      f crash failures:
(1a)  for round from 1 to f+1 do
(1b)    if the current value of x has not been broadcast then
(1c)        broadcast(x);
(1d)    y_j ← value (if any) received from process j in this round;
(1e)    x ← min_{v_j}(x, y_j);
(1f)  output x as the consensus value.
```

RAMAIAH
Institute of Technology

Consensus algorithms for Byzantine failures (synchronous system)

Impossibility of achieving Byzantine agreement with $n = 3$ processes and $f = 1$ malicious process.



system only if the number of Byzantine processes f is such that $f \leq \lceil \frac{n-1}{3} \rceil$ [20, 25].

We informally justify this result using two steps:

- With $n = 3$ processes, the Byzantine agreement problem cannot be solved if the number of Byzantine processes $f = 1$. The argument uses the illustration in Figure 14.3, which shows a commander P_c and two lieutenant processes P_a and P_b. The malicious process is the lieutenant P_b in the first scenario (Figure 14.3(a)) and hence P_a should agree on the value of the loyal commander P_c, which is 0. But note the second scenario (Figure 14.3(b)) in which P_a receives identical values from P_b and P_c, but now P_c is the disloyal commander whereas P_b is a loyal lieutenant. In this case, P_a needs to agree with P_b. However, P_a cannot distinguish between the two scenarios and any further message exchange does not help because each process has already conveyed what it knows from the third process.

  In both scenarios, P_a gets different values from the other two processes. In the first scenario, it needs to agree on a 0, and if that is the default value, the decision is correct, but then if it is in the second indistinguishable scenario, it agrees on an incorrect value. A similar argument shows that if 1 is the default value, then in the first scenario, P_a makes an incorrect decision. This shows the impossibility of agreement when $n = 3$ and $f = 1$.
- With n processes and $f \geq n/3$ processes, the Byzantine agreement problem cannot be solved. The correctness argument of this result can be shown using reduction. Let $Z(3, 1)$ denote the Byzantine agreement problem for parameters $n = 3$ and $f = 1$. Let $Z(n \leq 3f, f)$ denote the Byzantine agreement problem for parameters $n(\leq 3f)$ and f. A reduction from $Z(3, 1)$ to $Z(n \leq 3f, f)$ needs to be shown, i.e., if $Z(n \leq 3f, f)$ is solvable, then $Z(3, 1)$ is also solvable. After showing this reduction, we can argue that as $Z(3, 1)$ is not solvable, $Z(n \leq 3f, f)$ is also not solvable.

Agreement in asynchronous message-passing systems with failures

- The *critical step* is an event that occurs at a single process. However, other processes cannot tell apart the two scenarios in which this process has crashed, and in which this process is extremely slow. In both scenarios, the other processes can continue to wait forever and hence the processes may not reach a consensus value, remaining in bivalent state.
- The *critical step* occurs at two or more independent (i.e., not send–receive related) events at different processes. However, as independent events at different processes can occur in any permutation, the *critical step* is not well-defined and hence this possibility is not admissible.

Agreement in asynchronous message-passing systems with failures

The impossibility result is significant because it implies that all problems to which the agreement problem can be reduced are also not solvable in any asynchronous system in which crash failures may occur. As all real systems are prone to crash failures, this result has practical significance. We can show that all the problems, such as the following, requiring consensus are not solvable in the face of even a single crash failure:

- The leader election problem.
- The computation of a network-side global function using broadcast–convergecast flows.
- Terminating reliable broadcast.
- Atomic broadcast.

The common strategy is to use a reduction mapping from the consensus problem to the problem X under consideration. We need to show that, by using an algorithm to solve X, we can solve consensus. But as consensus is unsolvable, so must be problem X.

A predicate is stable if it does not turn false once it becomes true.

Thank you