CSE535 Asynchronous Systems
Termination Detection 2

YoungMin Kwon

SUNY Korea
The State University of New York

Termination Detection in General Distributed Computing Model

- A passive process does not necessarily become active on the receipt of a message
- Activation condition
  - A passive process requires a set of messages to become active

SUNY Korea
The State University of New York

Model Definition

- AND model
  - A passive process P_i can be activated after a message from every process in DS_i has arrived
- OR model
  - A passive process P_i can be activated after a message from any process in DS_i has arrived
- AND-OR model
  - A passive process P_i can be activated after a message from every process in $DS_i^{q_i}$ for some q_i has arrived
- k out of n model
  - A passive process P_i can be activated after a message from k_i distinctive processes in DS_i has arrived
- Predicate fulfilled
  - $fulfilled_i(A)$ is true iff messages arrived (not yet consumed) from all processes belong to set A are sufficient to activate P_i

SUNY Korea

Notations

- $passive_i$: true iff p_i is passive
- $empty(j,i)$: true iff all messages sent by p_j to p_i have arrived at p_i (either consumed or in the local buffer)
- $arr_i(j)$: true iff a message from p_j to p_i has arrived at p_i and not yet be consumed by p_i
- ARR_i: the set $\{ p_j : arr_i(j) \}$
- NE_i: the set $\{ p_j : \text{not } empty(j,i) \}$

SUNY Korea

Termination Definitions

- Dynamic Termination

  $$Dterm \equiv \forall P_i \in P : passive_i \wedge \neg fulfilled_i(ARR_i \cup NE_i)$$

  - No more activity is possible although messages can still be in transit

- Static Termination

  $$Sterm \equiv \forall P_i \in P : passive_i \wedge (NE_i = \emptyset) \wedge \neg fulfilled_i(ARR_i)$$

  - All channels are empty and none of the processes can be activated

SUNY Korea

Static Termination Detection Algorithm

- C_i, called a controller, is associated with each process p_i
- C_α initiates the detection by sending a query message to all controllers
- C_i responds with a message reply(ld_i), and C_α decides the termination as $td := \bigwedge_{1 \leq i \leq n} ld_i$
- If *td* is false C_α sends new query messages (new wave)

SUNY Korea
The State University of New York

Static Termination Detection Algorithm

- $ld_1, \ldots, ld_n$ must satisfy

$$\bigwedge_{1 \le i \le n} ld_i \Longrightarrow Sterm$$

$$\Longrightarrow \forall P_i \in P : passive_i \wedge (NE_i = \emptyset) \wedge \neg fulfilled_i(ARR_i)$$

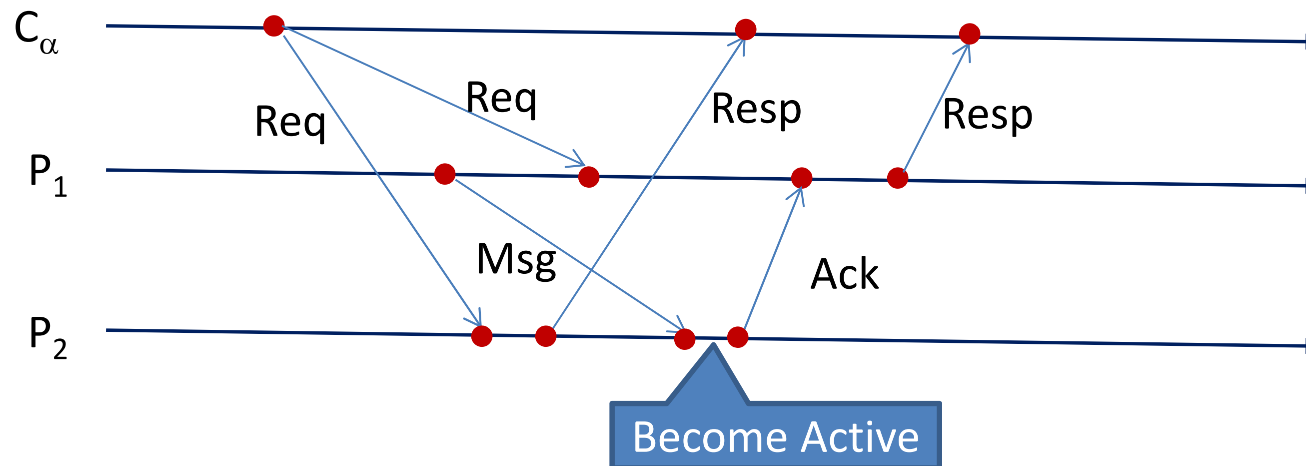
- C_i delays the response to a query as long as the following is false

$$passive_i \wedge (notack_i = 0) \wedge \neg fulfilled_i(ARR_i)$$

SUNY Korea

Static Termination Detection Algorithm

- Activity in the back of the wave



SUNY Korea

Static Termination Detection Algorithm

- Detecting the activity in the back of the wave
  - C_i maintains a variable cp_i
  - cp_i is initialized to true iff p_i is initially passive
  - When p_i becomes active, cp_i is set to false
  - When C_i sends a reply to C_α, cp_i is sent with the message and cp_i is set to true

- When cp_i is sent to C_α as true,
  - p_i has been passive since the previous wave
  - Messages arrived and not yet consumed are not sufficient to activate p_i
  - All output channels of p_i are empty

SUNY Korea

Static Termination Detection Algorithm

S1: When P_i sends a message to P_j

$$notack_i := notack_i + 1$$

S2: When a message from P_j arrives to P_i

send *ack* to C_j

S3: When C_i receives *ack* from C_j

$$notack_i = notack_i - 1$$

S4: When P_i becomes active

$$cp_i := false.$$

SUNY Korea

Static Termination Detection Algorithm

S5: When C_i receives query from C_α

Wait until

$((passive_i \wedge (notack_i = \emptyset) \wedge \neg fulfilled_i(ARR_i));$

$ld_i := cp_i;$

$cp_i := true;$

send $reply(ld_i)$ to C_α

S6: When controller C_a decides to detect static termination

repeat send $query$ to all C_i;

receive $reply(ld_i)$ from all C_i;

$$td := \bigwedge_{1 \leq i \leq n} ld_i;$$

$until$ td;

claim static termination

SUNY Korea

Dynamic Termination Detection Algorithm

- In static termination detection, there are no transit messages on a channel (using ack) and each controller knows the messages arrived.

- Dynamic termination detection can occur before all messages of the computation have arrived.

SUNY Korea
The State University of New York

Dynamic Termination Detection Algorithm

- In addition to cp_i, each C_i has two vector variables
  - $s_i[j]$: number of messages sent from p_i to p_j
  - $r_i[j]$: number of messages received by p_i from p_j

- C_α maintains an *nxn* matrix S, where
  - S[i,j]: C_α's knowledge about the number of messages sent by p_i to p_j

SUNY Korea

Dynamic Termination Detection Algorithm

- C_α sends to each C_i a query containing the vector (S[1, i] … S[n, i]) denoted by S[·, i]
- On receiving the query C_i approximately computes the set of non-empty channels $ANE_i = \{ p_j : s[j,i] - r_i[j] > 0 \}$
  - The approximate knowledge is sufficient to ensure the correctness
  - s[j,i] is accurate if p_j has been passive since the previous wave
- C_i computes ld_i which is true iff
  - p_i has been passive since the last wave
  - The activation cannot be fulfilled by ARR_i and ANE_i
- C_i sends to C_α a reply with ld_i and s_i
  - C_α updates the row S[i, ·] with s_i

SUNY Korea
The State University of New York

Dynamic Termination Detection Algorithm

S1: When P_i sends a message to P_j

$$s_i[j] := s_i[j] + 1$$

S2: When a message from P_j arrives at P_i

$$r_i[j] := r_i[j] + 1$$

S3: When P_i becomes active

$$cp_i := false$$

SUNY Korea
The State University of New York

Dynamic Termination Detection Algorithm

S4: When C_i receives $query(VC[1...n])$ from C_α

$(^*VC[1...n] = S[1...n, i]$ is the ith column of $S^*)$

$ANE_i := \{P_j : VC[j] > r_i[j]\};$

$ld_i := cp_i \wedge \neg fulfilled_i(ARR_i \cup ANE_i);$

$cp_i := (state_i = \text{passive});$

send $reply(ld_i, s_i)$ to C_α

S5: When controller C_α decides to detect dynamic termination

repeat for each C_i

send $query(S[1\ldots n, i])$ to C_i;

$(^*$ the ith column of S is sent to C_i $^*)$

receive $reply(ld_i, s_i)$ from all C_i;

$\forall i \in [1..n] : S[i, .] := s_i;$

$$\text{td} := \bigwedge_{1 \le i \le n} ld_i$$

until td;

claim dynamic termination

SUNY Korea

Termination Detection in Atomic Computation Model

- Atomic model of execution
  - A process may at any time take messages,
  - instantly change its internal state, and
  - at the same time send out zero or more message immediately
- All local actions at a process are performed in zero time
  - No need to consider process states
  - A distributed computation terminates at time t if all communication channels are empty at this instant

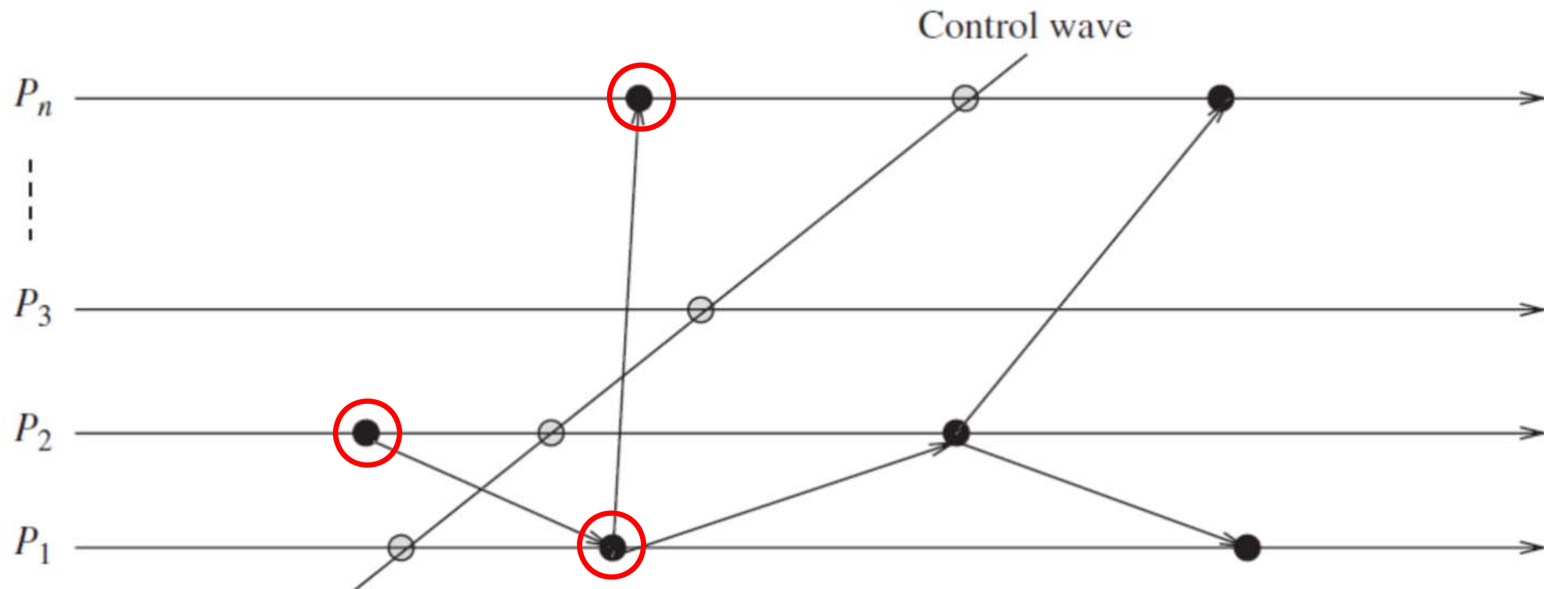
SUNY Korea

A Naive Counting Method

- Find out if there are any messages in transit
  - Every process p_i counts every message sent (denoted by $s_i(t)$) and received (denoted by $r_i(t)$)
  - Upon request, send the counts to the initiator
  - If the sum of s_i (t) for all p_i equals the sum of r_i (t) for all p_i, there are no messages in transit

SUNY Korea
The State University of New York

A Naive Counting Method



- Incorrect result
  - Send in P_2 is counted, but receive in P_1 is not counted
  - Send in P_1 is not counted, but receive in P_n is counted.

SUNY Korea

The Four Counter Method

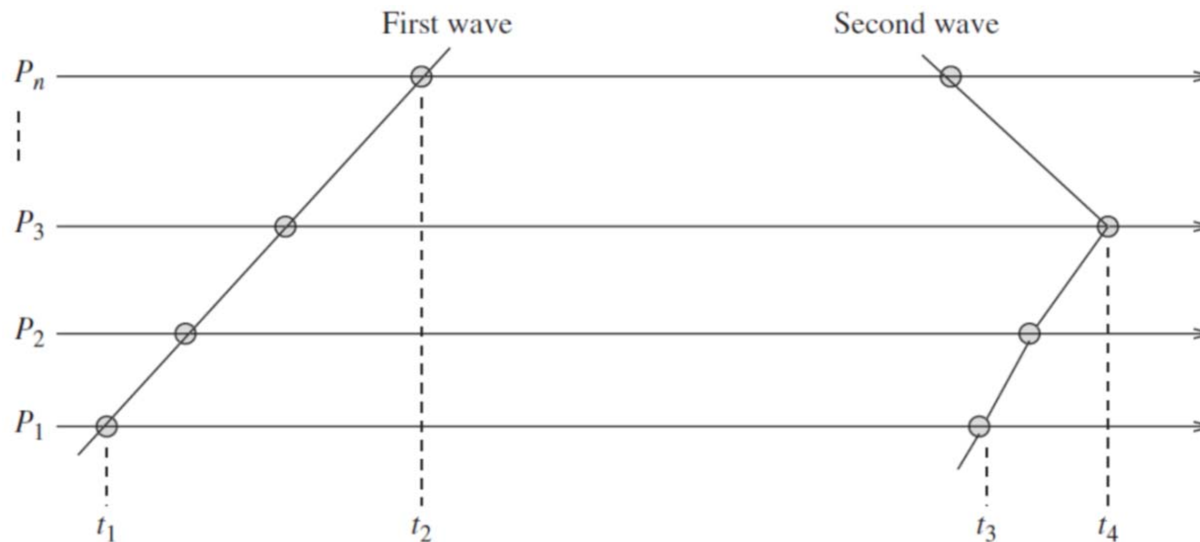
- Count twice using the naive counting method and compare the results
- The initiator accumulated the counts in the response as

$$R^* := \sum_{\forall i} r_i(t_i) \text{ and } S^* := \sum_{\forall i} s_i(t_i)$$

- It starts the second wave and get R'* and S'*
- If R* = S* = R'* = S'* the system is terminated
  - If R* = S'* then the system is terminated at the end of the first wave

SUNY Korea

The Four Counter Method



- t_2 : the time when the first wave is finished
- t_3 ($\geq t_2$) : the starting time of the second wave

SUNY Korea
The State University of New York

The Four Counter Method

1. Local message counters are monotonic
   - $t \leq t'$ implies $s_i(t) \leq s_i(t')$ and $r_i(t) \leq r_i(t')$
2. The total number of messages sent and received are monotonic
   - $t \leq t'$ implies $S(t) \leq S(t')$ and $R(t) \leq R(t')$
3. $R^* \leq R(t_2)$: because of (1) and all values r_i are collected before t_2
4. $S'^* \geq S(t_3)$: because of (1) and all values s_i are collected after t_3
5. For all t, $R(t) \leq S(t)$. The number of messages in transit is $D(t) := S(t) - R(t) \geq 0$

SUNY Korea

The Four Counter Method

- Now, we can show that if $R^* = S'^*$, then the computation had terminated at the end of the first wave.

$$R^* = S'^* \Longrightarrow R(t_2) \geq S(t_3)$$
$$\Longrightarrow R(t_2) \geq S(t_2)$$
$$\Longrightarrow R(t_2) = S(t_2)$$

- That is, the channel is empty and the computation is terminated at t_2

SUNY Korea

The Sceptic Algorithm

- Four counter method
  - The counter values obtained by the first wave become corrupted if there is some activity at the right of the wave
- To detect such activities use flags
  - At the first wave, initialize the flag
  - After sending/receiving a message set the flag
  - Second wave checks if there is any flag set

SUNY Korea

The Time Algorithm

- Four counter method and Sceptic algorithm require at least two waves.
  - The time algorithm is a single wave detection algorithm
  - Each process has a local clock
- The role of a local clock
  - Control messages are sent along a ring structure
  - Local clock (the epoch of the latest wave) is piggybacked in the message
  - On receiving a control message, an activity in the back of the wave can be caught by comparing the local clock and the control message

SUNY Korea

The Time Algorithm

- Algorithm
  - The initiator starts the control wave at time i, accumulates a send/receive counter and synchronizes the local clock by setting them to i+1
  - The control wave distinguishes the past and the future
  - If a process receives a message whose timestamp is greater than its local time, nullify the wave

SUNY Korea

The Time Algorithm for P_j

(a) When sending a basic message to P_i:

(1) $COUNT \leftarrow COUNT + 1$;

(2) **send** <*CLOCK*,...> to P_i;

/* timestamped basic message */

(b) When receiving a basic message <*TSTAMP*,...>:

(3) $COUNT \leftarrow COUNT - 1$;

(4) $TMAX \leftarrow max(TSTAMP, TMAX)$;

(5) /* process the message */

(c) When receiving a control message <*TIME*, *ACCU*, *INVALID*, *INIT*>:

(6) $CLOCK \leftarrow max(TIME, CLOCK)$: /* synchronize the local closk */

(7) **if** $INIT = j$ /* complete round? */

(8) **then if** $ACCU = 0$ and not *INVALID*

(9) **then** "terminated" **else** "try again";

(10) **endif**;

(11) **else send** <*TIME*, *ACCU* + *COUNT*, *INVALID* or *TMAX* ≥ *TIME*, *INIT*> **to** $P_{(j \bmod n)+1}$;

(12) **end_if**;

(d) When starting a control round:

(13) $CLOCK \leftarrow CLOCK + 1$;

(14) **send** <*CLOCK*, *COUNT*, *false*, *j*> to $P_{(j \bmod n)+1}$;

SUNY Korea
The State University of New York