Week 8: Distributed Transactions
Part 4: Deadlock
Deadlock

Four conditions for deadlock

1. Mutual exclusion
   Transactions get exclusive locks on resources

2. Hold and wait
   A lock isn’t released but we wait for another

3. Non-preemption
   A transaction cannot access a resource another locked

4. Circular wait
   There’s a circular dependency of transactions waiting on locked resources
Resource $R_1$ is allocated to process $P_1$

Resource $R_1$ is requested by process $P_2$

This is called a Wait-For Graph (WFG)

Deadlock is present when the graph has cycles
A circular dependency among four processes and four resources leads to deadlock.
Dealing with deadlock

Same conditions for distributed systems as centralized
Harder to detect, avoid, prevent

Strategies

1. **Ignore**
   Do nothing. So easy & so tempting.

2. **Detect**
   Allow the deadlock to occur, detect it, and then deal with it by aborting and restarting a transaction that causes deadlock.

3. **Prevent**
   Make deadlock impossible by granting requests such that one of the conditions necessary for deadlock does not hold.

4. **Avoid**
   Choose resource allocation so deadlock does not occur. *But the algorithm needs to know what resources will be used and when → not feasible in most cases*
Deadlock detection

Kill off a task when deadlock is detected
  – That will break the circular dependency

• It might not feel good to kill a process...
  – But transactions are designed to be abortable

• So, just **abort** the transaction
  – Data is restored to its state before the transaction began
  – The transaction can restart at a later time
  – Resource allocation in the system may be different in the future, so the transaction may succeed the next time it's run
Centralized deadlock detection

Imitate the non-distributed algorithm through a coordinator

• Each system maintains a **Wait-For Graph** for its processes and resources

• A **central coordinator** maintains the combined graph for the entire system: the **Global Wait-For Graph**
  – A message is sent to the coordinator each time an edge (resource hold/request) is added or deleted
  – List of adds/deletes can be sent periodically
Centralized deadlock detection

Local Wait-For Graph on A

Local Wait-For Graph on B

Global Wait-For Graph
Centralized deadlock detection

Two events occur:
1. Process $P_2$ releases resource $T$ on system $B$
2. Process $P_1$ asks system $B$ for resource $T$

Two messages are sent to the coordinator:
Message 1 (from B): $P_2$ releases $T$
Message 2 (from A): $P_1$ waits for $T$

If message 2 arrives first, the coordinator constructs a graph that has a cycle and hence detects a deadlock

This is **phantom deadlock**

A phantom deadlock is known as a false deadlock
Example: No Phantom Deadlock

No deadlock

Message 1 from B: 
 release(T)

Message 2 from A: 
 wait_for(T)

All good: no deadlock detected!
Phantom Deadlock Example

No deadlock

DEADLOCK detected!

We detected deadlock because the coordinator received the messages out of order.

It really wasn’t deadlock since P₂ released T Too Late!
Avoiding Phantom Deadlock

- Impose globally consistent (total) ordering on all processes

  or

- Have coordinator reliably ask each process whether it has any release messages
Distributed deadlock detection

- Processes can request multiple resources at once
  - Consequence: process may wait on multiple resources

- Some processes wait for local resources

- Some processes wait for resources on other machines

- Algorithm invoked when a process has to wait for a resource
Distributed deadlock detection algorithm

Chandy-Misra-Haas algorithm

Edge Chasing

When requesting a resource, generate a probe message

– Send to all process(es) currently holding the needed resource
– Message contains three process IDs: \{ blocked\_ID, my\_ID, holder\_ID \}

1. Process that originated the message (blocked\_ID)
2. Process sending (or forwarding) the message (my\_ID)
3. Process to whom the message is being sent (holder\_ID)
Chandy-Misra-Haas algorithm

If a process receives a probe message:

• Check to see if it is waiting for any resources held by other processes

• For each process holding a resource it is waiting for:
  – Update & forward a probe message: \{\textit{blocked\_ID}, \textit{my\_ID}, \textit{holder\_ID}\}
    • Replace \textit{my\_ID} field by its own process ID
    • Replace \textit{holder\_ID} field by the ID of the process it is waiting for
    • Send messages to each process on which it is blocked

If a message goes all the way around and comes back to the original sender, a cycle exists ⇒ \textit{we have deadlock}
Chandy-Misra-Haas algorithm – edge chasing

- Process 0 needs a resource process 1 is holding
- That means process 0 will block on process 1
  - Send initial message from P0 to P1: (0,0,1)
  - P1 sends (0, 1, 2) to P2; P2 sends (0, 2, 3) to P3
- Message (0,8,0) returns back to sender
  ⇒ Cycle exists: we will have deadlock if P_0 blocks on the resource
Distributed deadlock prevention

Design the system so that deadlocks are structurally impossible

Disallow at least one of the four conditions for deadlock:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mutual exclusion</strong></td>
<td>• Allow a resource to be held (used) by more than one process at a time</td>
</tr>
<tr>
<td></td>
<td>• <strong>Not practical</strong> if an object gets modified.</td>
</tr>
<tr>
<td></td>
<td>• This can violate the ACID properties of a transaction</td>
</tr>
<tr>
<td><strong>Non-preemption</strong></td>
<td>• Essentially gives up mutual exclusion</td>
</tr>
<tr>
<td></td>
<td>• This can also violate the ACID properties</td>
</tr>
<tr>
<td></td>
<td>• We can use optimistic concurrency control algorithms and check for conflicts at commit time and roll back if needed</td>
</tr>
<tr>
<td><strong>Hold and wait</strong></td>
<td>• Implies that a process gets all its resources at once</td>
</tr>
<tr>
<td></td>
<td>• Not practical to disallow this – we don’t know what resources a process will use</td>
</tr>
<tr>
<td><strong>Circular wait</strong></td>
<td>• Ensure that a cycle of waiting on resources does not occur</td>
</tr>
</tbody>
</table>
Distributed deadlock prevention

Deny circular wait

• Assign a unique timestamp to each transaction

• Ensure that the Global Wait-For Graph can only proceed from young to old or from old to young
Deadlock prevention: timestamp ordering

When a transaction is about to block waiting for a resource used by another, check to see which has a larger timestamp (which is older)

• Allow the wait only if the waiting transaction has a lower (older) timestamp than the transaction waited on

• Timestamps in a resource allocation graph must always increase, so cycles are impossible

• Alternatively: allow transactions to wait only if the waiting transaction has a higher (younger) timestamp than the transaction it's waiting on
Wait-die algorithm

• Old process wants resource held by a younger process
  – Old process waits

• Young process wants resource held by older process
  – Young process kills itself

Only permit older processes to wait on resources held by younger processes
Wound-wait algorithm

- Kill the resource owner if needed
- Old process wants resource held by a younger process
  - Old process kills the younger process
- Young process wants resource held by older process
  - Young process waits

Only permit younger processes to wait on resources held by older processes
The End