고려대학교
KOREA UNIVERSITY

KU-The Future

Cloud computing – Distributed Systems

Heonchang Yu

Distributed and Cloud Computing Lab.

Distributed deadlocks

- **Deadlocks**
  - ✓ Can arise within a single server when locking is used for concurrency control
  - ✓ Servers must either prevent or detect and resolve **deadlocks.**
  - ✓ Using **timeouts** to resolve possible deadlocks is a clumsy approach.
  - ✓ Most **deadlock detection schemes** operate by finding cycles in the transaction wait-for graph.
    - ❖ Nodes: transactions and objects
    - ❖ Edges: an object held by a transaction or a transaction waiting for an object
    - ❖ **Distributed deadlock:** can be a cycle in the global wait-for graph that is not in any single local one
  - ✓ In figure 1(table) and figure 2(graph)
    - ❖ The objects A and B managed by servers X and Y and objects C and D managed by server Z
    - ❖ A deadlock cycle in figure 2(a)
    - ❖ Wait-for graph in figure 2(b)

Distributed deadlocks

U		*V*		*W*	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B*		
a.deposit(20)	lock *A*		at *Y*		
	at *X*				
				c.deposit(30)	lock *C*
b.withdraw(30)	wait at *Y*				at *Z*
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*

Figure 1 Interleaving of transaction *U*, *V* and *W*

Distributed deadlocks

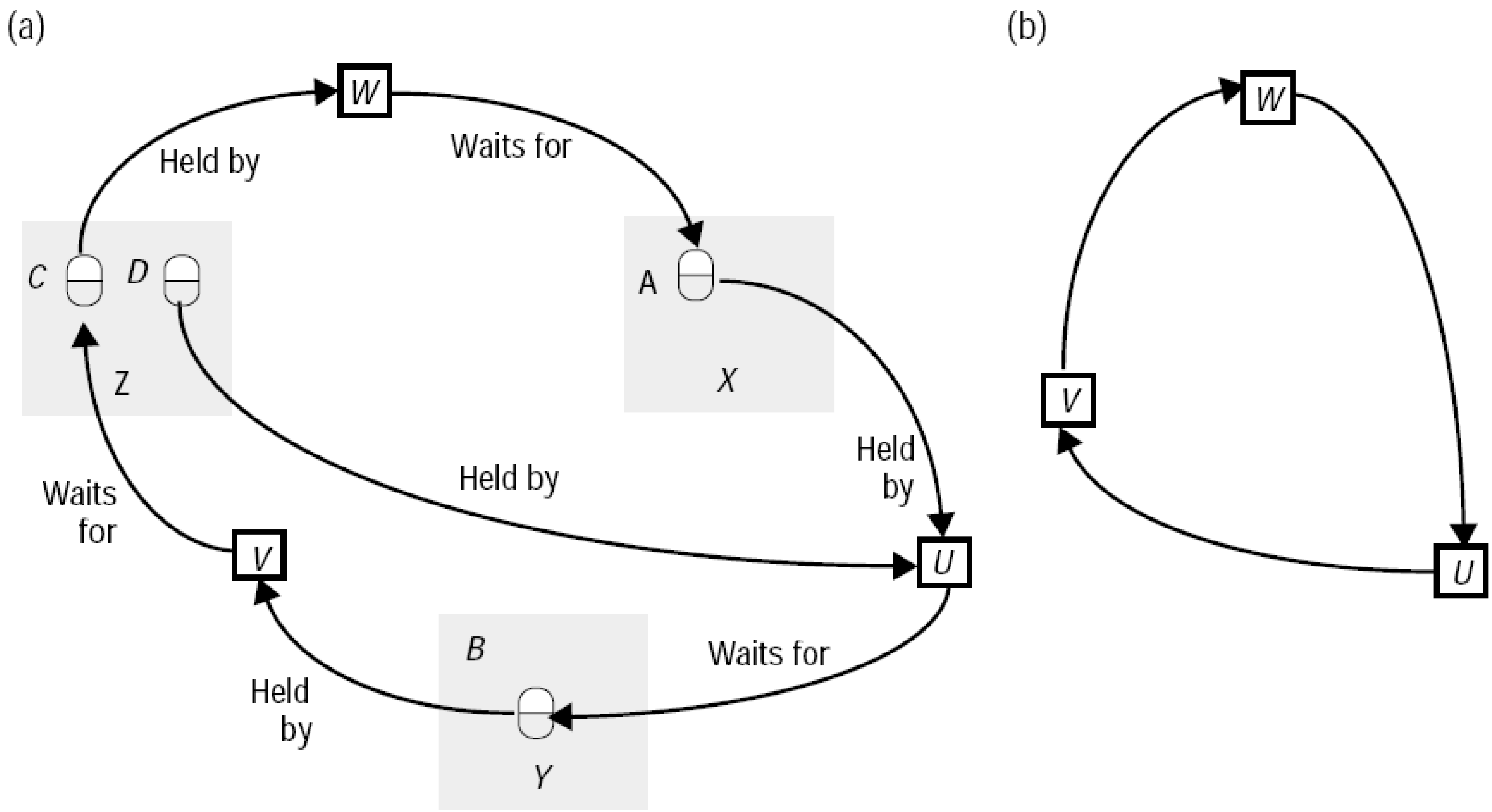

Figure 2 Distributed deadlock

Distributed deadlocks

- **Deadlocks**
  - ✓ Local wait-for graphs can be built by the lock manager at each server.
    - ❖ Server *Y*: ***U* → *V*** (added when *U* requests *b.withdraw*(30))
    - ❖ Server *Z*: ***V* → *W*** (added when *V* requests *c.withdraw*(20))
    - ❖ Server *X*: ***W* → *U*** (added when *W* requests *a.withdraw*(20))
  - ✓ **Centralized deadlock detection**
    - ❖ **One server** takes on the role of global deadlock detector.
    - ❖ Each server sends the latest copy of its **local wait-for graph** to the global deadlock detector.
    - ❖ When the detector finds a cycle, it **makes a decision** on how to resolve the deadlock and **informs the servers** as to the transaction to be aborted to resolve the deadlock.
    - ❖ Centralized deadlock detection is **not a good idea**, because it depends on a single server to carry it out.
    - ❖ **Problems:** poor availability, lack of fault tolerance, no ability to scale and the cost of the frequent transmission of local wait-for graph

Distributed deadlocks

- **Phantom deadlocks**
  - ✓ A deadlock that is 'detected' but is **not really a deadlock**
  - ✓ As the procedure for deadlock detection will **take some time,** there is a chance that one of the transactions that holds a lock will **meanwhile have released** it, in which case the deadlock will no longer exist.
  - ✓ In figure 3, it would detect a cycle $T \rightarrow U \rightarrow V \rightarrow T$, although the edge ***$T \rightarrow U$* no longer exists.**
    - ❖ If there is a cycle $T \rightarrow U \rightarrow V \rightarrow T$ and U aborts after the information concerning U has been collected, then the cycle has been already and there is no deadlock.

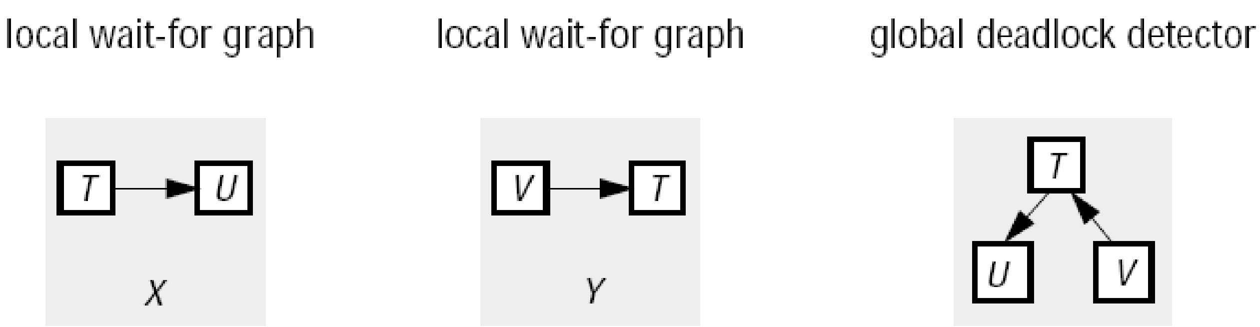

Figure 3 Local and global wait-for graphs

Distributed deadlocks

- **Edge chasing(=path pushing)**
  - ✓ The global wait-for graph is not constructed, but each of the servers involved has **knowledge** about some of its edges.
  - ✓ The servers attempt to find cycles by forwarding **messages called probes.**
  - ✓ A probe message consists of **transaction wait-for relationships** representing a path in the global wait-for graph.
  - ✓ The situation at server X in **figure 2**
    - ❖ The server X has just added **the edge $W \rightarrow U$** to its local wait-for graph and at this time, transaction **U is waiting to access object B**, which transaction V holds at server Y.
    - ❖ This edge could **possibly be part of a cycle** such as $V \rightarrow T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow W \rightarrow U \rightarrow V$ involving transactions using objects at other servers.
    - ❖ This indicates that there is **a potential distributed deadlock cycle**, which could be found by sending out a probe to server Y.

Distributed deadlocks

- **Edge chasing algorithm**
  - **Initiation:** When a server notes that a **transaction *T* starts waiting for another transaction *U***, where *U* is waiting to access an object at another server, it initiates detection by sending a probe containing the **edge $< T \rightarrow U >$** to the server of the object at which transaction *U* is blocked. If *U* is sharing a lock, probes are sent to all the holders of the lock.
  - **Detection:** Detection consists of **receiving** probes and **deciding** whether deadlock has occurred and whether to forward the probes.
    - When a server of an object receives a probe $< T \rightarrow U >$, it checks to see whether *U* is also waiting.
    - If it is, the transaction it waits for (for example, V) is added to the probe (making it $< T \rightarrow U \rightarrow V >$), and if the new transaction (*V*) is waiting for another object elsewhere, the probe is forwarded.
    - Before forwarding a probe, the server checks to see whether the transaction it has just added has caused the probe to contain a cycle (ex. $< T \rightarrow U \rightarrow V \rightarrow T >$). If this is the case, it has found a cycle in the graph and deadlock has been detected.
  - **Resolution:** When a cycle is detected, a transaction in the cycle is **aborted** <u>to break the deadlock</u>.

Distributed deadlocks

- **Edge chasing**
  - ✓ The example of figure 4
    - ❖ **Server *X*** initiates detection by sending **probe <*W*→*U*>** to the server of *B* (Server *Y*)
    - ❖ **Server *Y*** receives probe <*W*→*U* >, notes that *B* is held by *V* and appends *V* to the **probe to produce <*W*→*U*→*V*>**. It notes that *V* is waiting for *C* at server *Z*. This probe is forwarded to server *Z*.
    - ❖ **Server Z** receives probe <*W*→*U*→*V*> and notes *C* is held by *W* and appends *W* to the **probe to produce <*W*→*U*→*V*→*W*>**.
  - ✓ A probe that detects a cycle involving *N* transactions will be forwarded **by (*N*-1) transaction coordinators** via **(*N*-1) servers** of objects, requiring 2(*N*-1) messages.

Distributed deadlocks

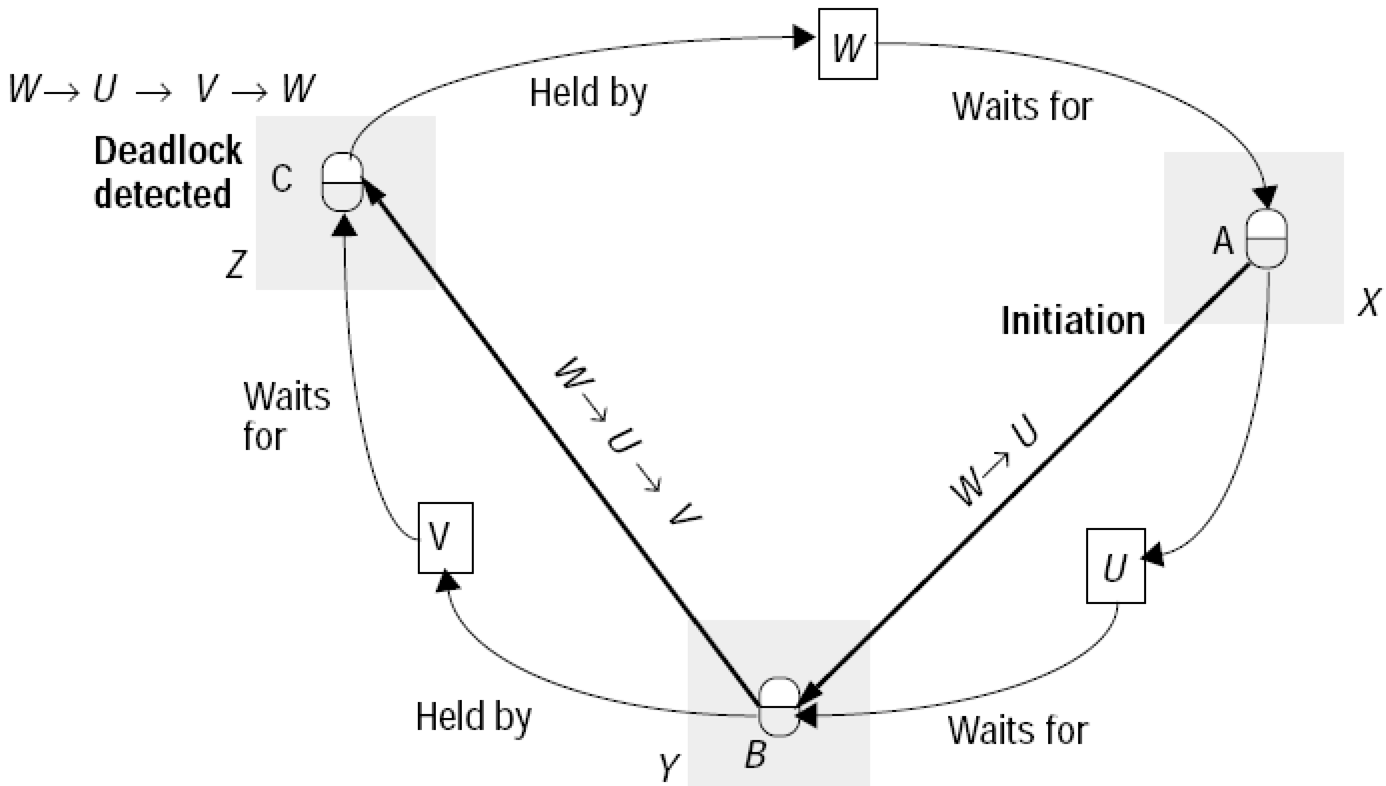

Figure 4 Probes transmitted to detect deadlock

Distributed deadlocks

- **Transaction priorities**
  - ✓ The effect of several transactions in a cycle initiating deadlock detection is that detection may **happen at several different servers** in the cycle with the result that <u>more than one transaction in the cycle is aborted.</u>
  - ✓ In figure 5
    - ❖ (a) : *U* is waiting for *W* and *V* is waiting for *T*. *T* requests the object held by *U* and *W* requests the object held by *V*. **Two separate probes $<T \rightarrow U>$ and $<W \rightarrow V>$** are initiated by the servers of these objects and are circulated until deadlock is detected by each of two different servers.
    - ❖ (b), (c) : Cycles are $<T \rightarrow U \rightarrow W \rightarrow V \rightarrow T>$ and $<W \rightarrow V \rightarrow T \rightarrow U \rightarrow W>$.
  - ✓ <u>In order to ensure that only one transaction in a cycle is aborted,</u> transactions are given **priorities** in such a way that all transactions are totally ordered. **(ex. Timestamps)**
    - ❖ When a deadlock cycle is found, the transaction with the **lowest priority is aborted.**
    - ❖ In the example of figure 5, **assume $T > U > V > W$.** <u>Then the transaction *W* will be aborted when a cycle is detected.</u>

Distributed deadlocks

- **Transaction priorities**
  - ✓ Transaction priorities could also be used **to reduce** the number of situations that cause deadlock detection to be initiated.
  - ✓ Transaction priorities could also be used **to reduce** the number of probes that are forwarded.
    - ❖ Probes should travel **'downhill'** – that is, from transactions with higher priorities to transactions with lower priorities.
    - ❖ Servers use the rule that they **do not forward** any probe to a holder that has higher priority than the initiator.

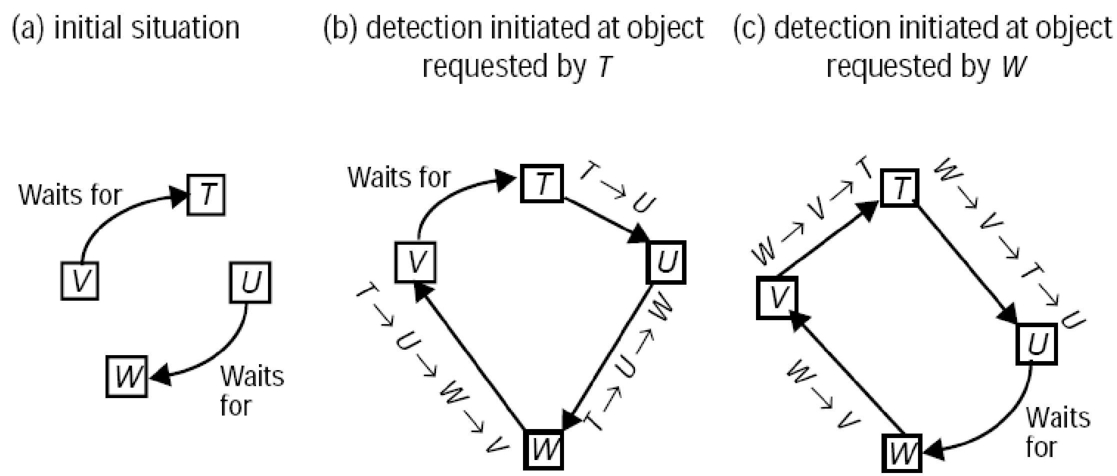

Figure 5 Two probes initiated

Distributed deadlocks

- ✓ Without priority rules, detection is initiated when *W* starts waiting by sending a probe <*W*→*U*>.
- ✓ Under the priority rule, this probe will not be sent, because *W*<*U* and deadlock will not be detected.
- ✓ In figure 6
  - ❖ (a) : When *U* starts waiting for *V*, the coordinator of *V* will save the probe <*U*→*V*>
  - ❖ (b) : When *V* starts waiting for *W*, the coordinator of *W* will store <*V*→ *W*> and *V* will forward its probe queue, <*U*→*V*>, to W.
  - ❖ *W* 's probe queue has <*U*→*V*> and <*V*→*W*>. When *W* starts waiting for *A* it will forward its probe queue <*U*→*V*→*W*> to the server of A, which also notes the new dependency *W*→*U* and combines it with the information in the probe received to determine that *U*→*V*→*W*→*U*. Deadlock is detected.

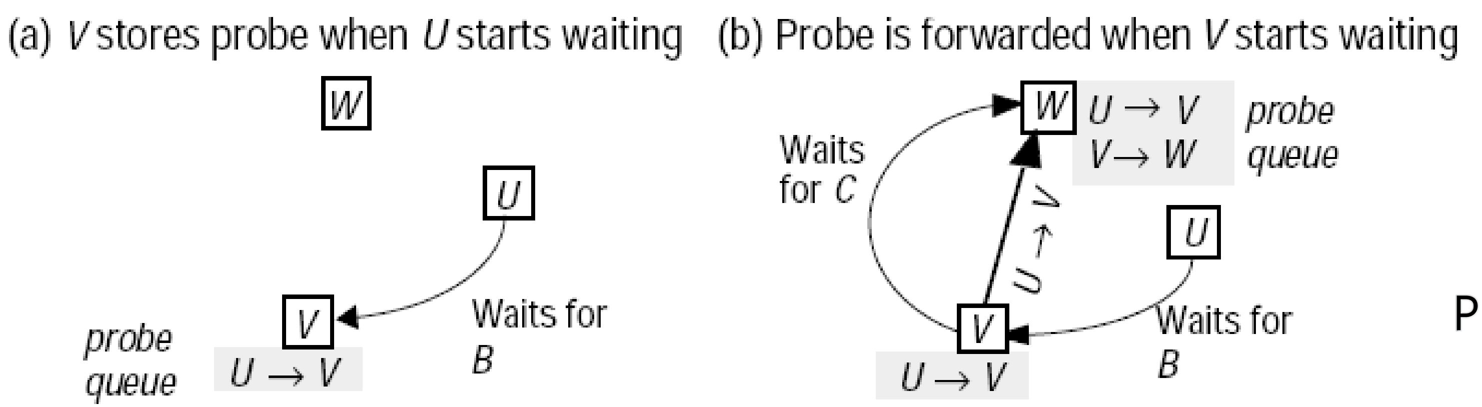

Figure 6
Probes travel downhill