Distributed Deadlocks

CS3524 Distributed Systems

Lecture 13

Deadlock

- A deadlock is a condition in a system, where a process (transaction) cannot proceed because it needs to obtain a resource (some data object) held by another process
- Processes may lock resources (see locking schemes) to exclusively reserve a resource for use
- Two elements
  - The “process” (or transaction): is executing actions and using resources
  - The “resource”: is used by processes during their execution (e.g. shared data), usually, a process puts a “lock” on a resource (see locking schemes)

Deadlock - Coffman Conditions

- A deadlock occurs under the following conditions
  - Mutual exclusion:
    - A resource is held by at most one process – this is a necessary condition to guarantee consistency in transaction management to serialise access to resources, but also creates the danger of deadlocks (see ACID principles)
  - Hold and Wait:
    - Processes that already hold resources, can wait for another resource to become available
  - Non-preemption:
    - A resource, once granted to a process, cannot be taken away – this is a necessary condition to guarantee consistency in transaction management, but also creates the danger of deadlock (see ACID principles and 2-phase locking)
  - Circular wait:
    - Two or more processes mutually wait for resources to become available that each of them holds and the other processes want – one process waits forever for another process to release its resource

Example: Deadlock with Write Locks

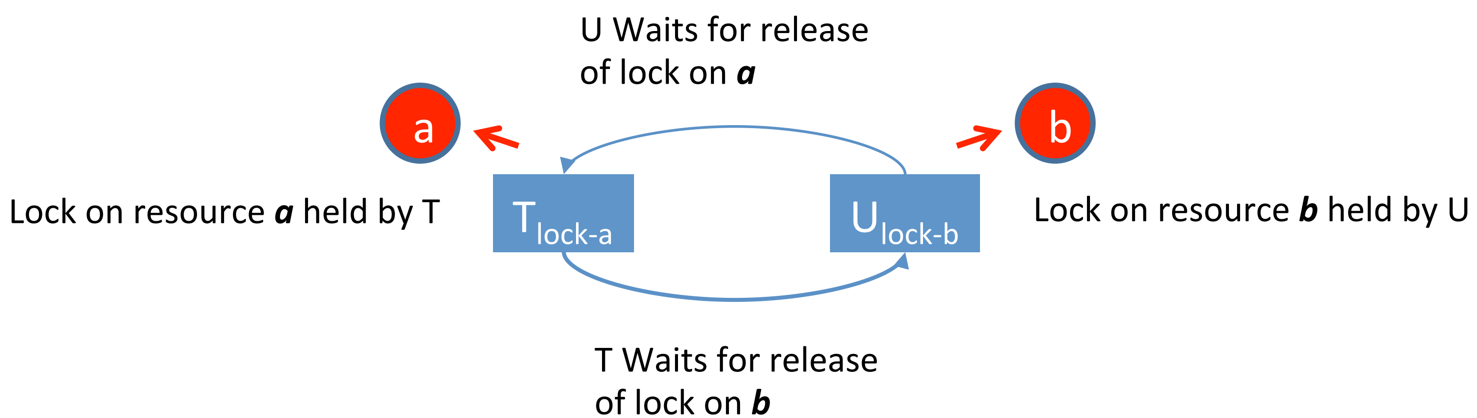
Transaction T		Transaction U	
a.deposit (100)	Acquire write lock on a		
		b.deposit (200)	Acquire write lock on b
b.withdraw (100)	Wait for lock on b to be released by U		
		a.withdraw (200)	Wait for lock on a to be released by T
Wait ….. Wait …..			
		Wait ….. Wait …..	

Deadlock Resolution

- Deadlock detection and resolution
  - Deadlocks have to be detected and resolved by a transaction management system (transaction scheduler):
  - Deadlocks can be “broken” by simply aborting one of the transactions
- But: which one ??
- Possible criteria for choosing such a transaction:
  - Put a time stamp on each transaction and abort the oldest one
  - Abort the “most complex” transaction (with many resources locked)
- Deadlock resolution without detection – e.g. Use time-outs:
  - Transactions operate with time-outs
  - But: how long should a time-out period be??

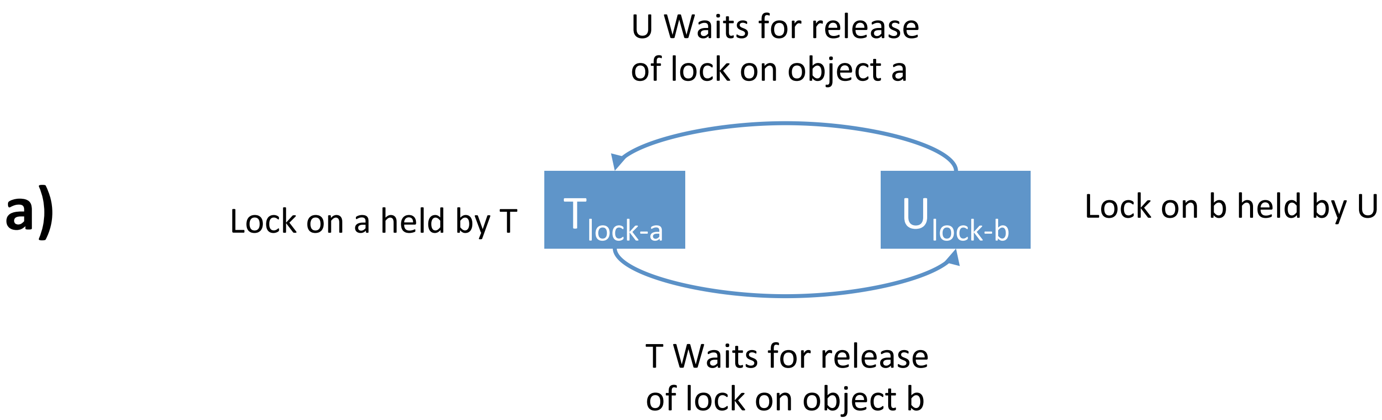
Deadlock Detection: Wait-for Graph

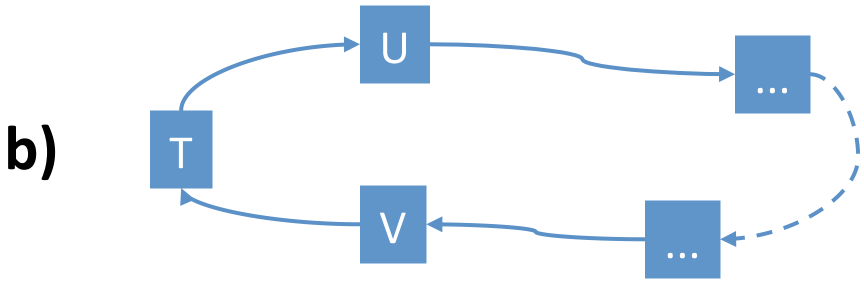
- Representation of a deadlock situation with a Wait-for graph:
  - Nodes in this graph represent transactions
  - Edges between nodes represent wait-for relationships between current transactions
  - The dependency between transactions is indirect – via a dependency on objects
  - A cycle in the graph indicates a deadlock:



  - Transaction T and U wait for each other because of a lock on one object
  - None of these locks can ever be released
  - One of the transactions would have to be aborted to break this deadlock

Wait-for Graph

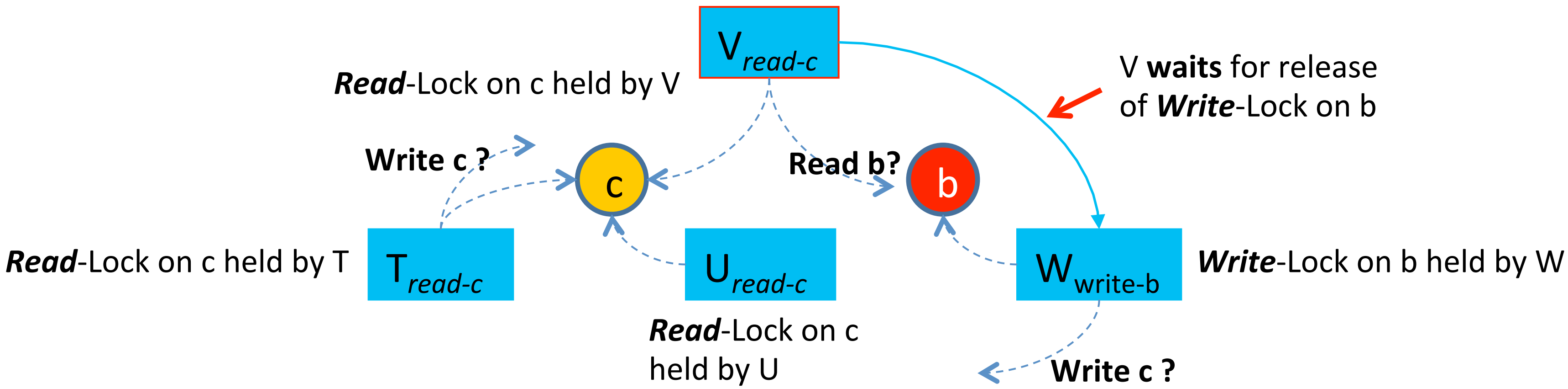




- Note: in case b), a cycle T --> U --> ... --> V --> T exists – all these transactions are blocked waiting for locks

Deadlock with Read and Write Locks

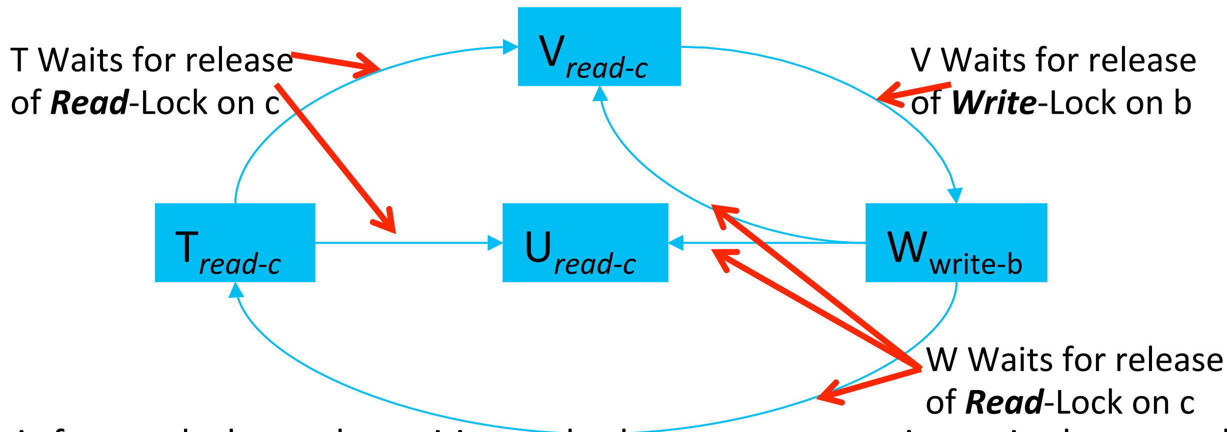
Transaction V tries to obtain a *Read-Lock* on object b



- Transactions T, U, V share a **Read-Lock** on object c, transaction W holds a **Write-Lock** on b
- Waiting transactions:
  - Transaction V tries to obtain a **Read-Lock** on b, waits for the Write-Lock to be released by W (see wait-for graph)
  - Can W release the Write-lock ?
- We assume the following scenario:
  - W wants a Write-Lock on c, T wants a Write-Lock on c

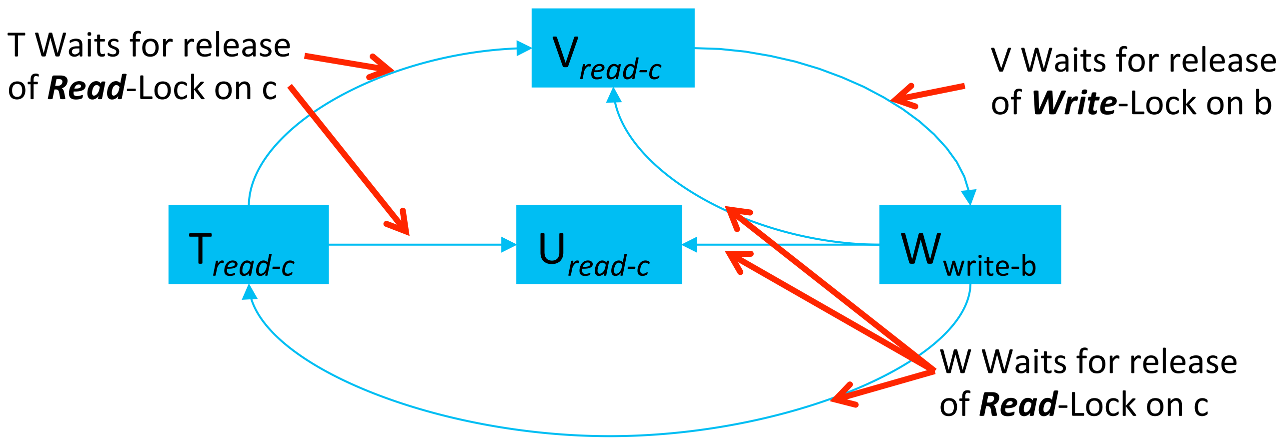
Deadlock with Read and Write Locks

- T, U, V share a read lock on object c, W holds a write lock on object b
- Scenario: Transactions T and W request a write lock on object c
  - A dead lock arises: T cannot promote its lock on c to a Write-Lock, because U and V still hold a Read-Lock: T waits-for U,V
  - V is waiting to obtain a read lock on object b, waits for W to release write lock – DEADLOCK – W is waiting for V to release its Read-Lock
  - W waits to obtain write lock on object c, waits for T, U, V to release read lock, W cannot set a Write-Lock on object c, because T,U,V hold read locks
  - T waits to obtain write lock on object c, waits for U, V to release read lock



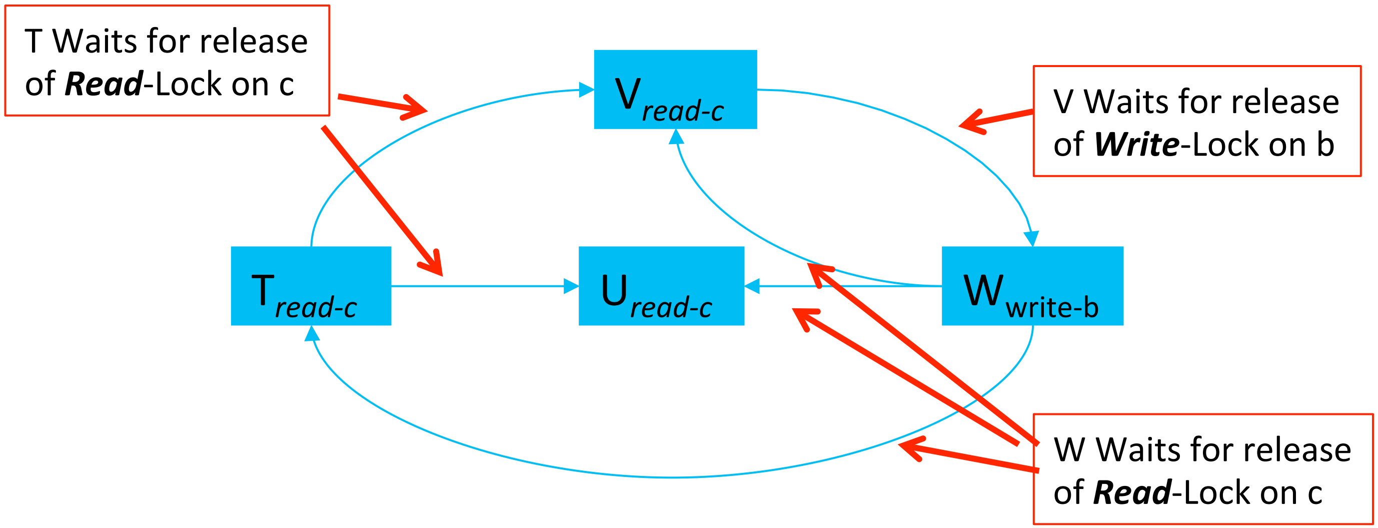
  - Wait-for graph shows the waiting cycles between transactions – in the example, there are two of them: <V --> W -->T-->V> and <V-->W-->V>

Deadlock with Read and Write Locks



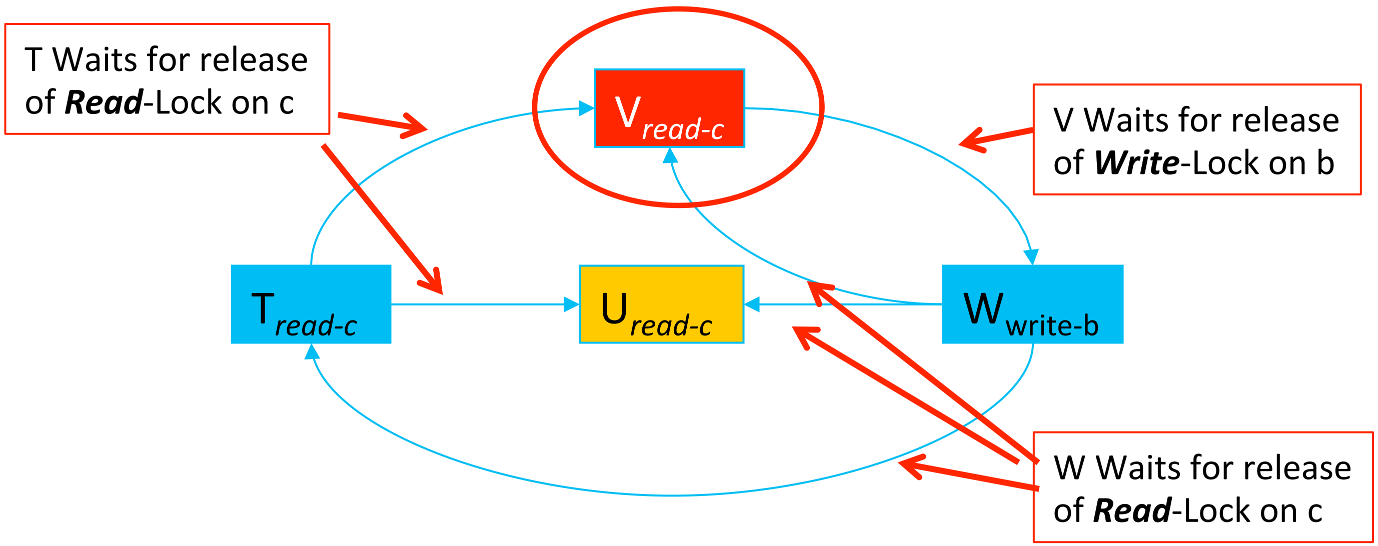
- T, U, V share a read lock on object c, W holds a write lock on object b
- Scenario:
  - Transaction V requests a read-lock on object b
    - Has to wait for Transaction W to release write-lock on b
  - Transactions W requests a write-lock on object c
    - Have to wait for Transactions T,U,V to release read-locks on c
  - Transaction T wants to promote its own read-lock on c to a write-lock
    - Has to wait for transactions U and V to release read-locks on c

Deadlock with Read and Write Locks



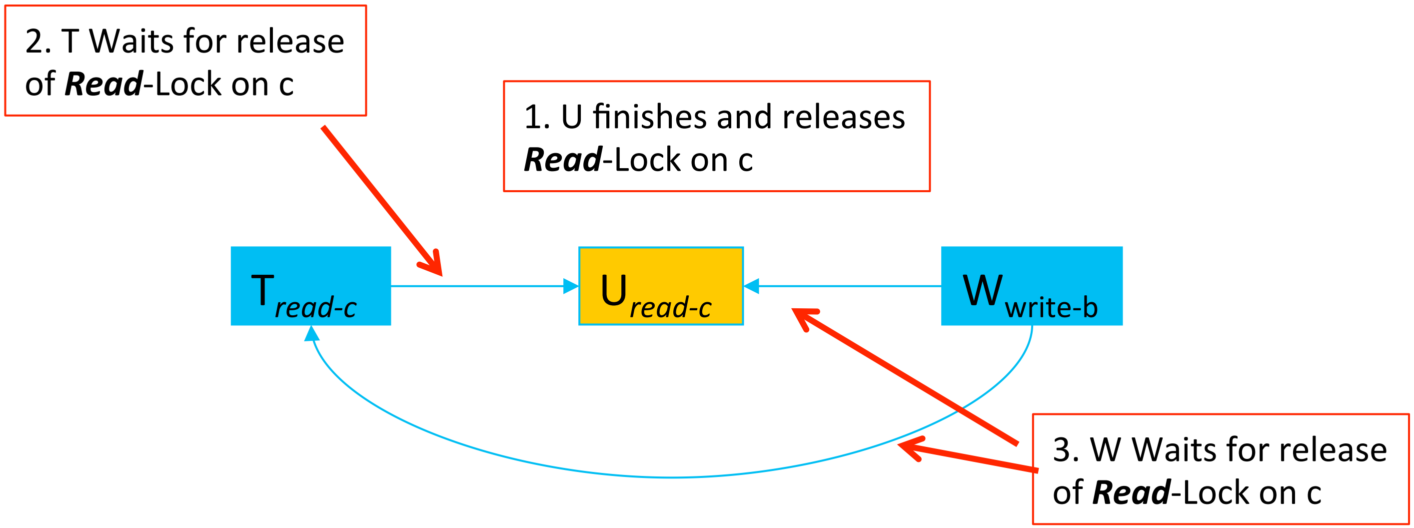
- Deadlock situations
  - T cannot promote its lock on c to a Write-Lock, because U and V still hold Read-Locks: T waits for U,V to release Read-locks on c
  - V waits for W to release Write-lock on b
  - W cannot set a Write-Lock on object c, because T,U and V hold read locks: W waits for T, U and V to release Read-lock on c
- Wait-for graph shows two waiting cycles:
  - <V --> W -->T-->V> and <V-->W-->V>

Break Deadlock



- Wait-for Graph shows how to break this deadlock
  - V is in both waiting cycles
  - If V is **aborted**, then its Read-lock on object c is released
  - **U is not waiting for anything**, therefore we assume that it **ends naturally** and releases the read lock on object c
  - T can now promote its Read-lock on c to a Write-lock, it can now perform its write(), end naturally and release write lock on c
  - W can now obtain the write lock on object c

Break Deadlock



- Abort transaction V
- The rest of the transactions execute in the following sequence
  - **U is not waiting for anything**, it **ends naturally** and releases the read lock on object c
  - T can now promote its Read-lock on c to a Write-lock, it can now perform its write(), end naturally and release write lock on c
  - W can now obtain the write lock on object c and end naturally

Handling Deadlocks

- Deadlock Detection
  - Use the Wait-for graph to identify cycles and select transactions to be aborted
    - Select the oldest transactions
    - Select the transaction involved in most of the cycles
    - Select according to their complexity
- Deadlock Prevention
  - Lock all objects at the very beginning of a transaction in one atomic action
  - Problem
    - Reduced concurrency: unnecessary access restriction to shared resources
    - It must be know in advance which objects are manipulated --> this is impossible in interactive applications

Handling Deadlocks

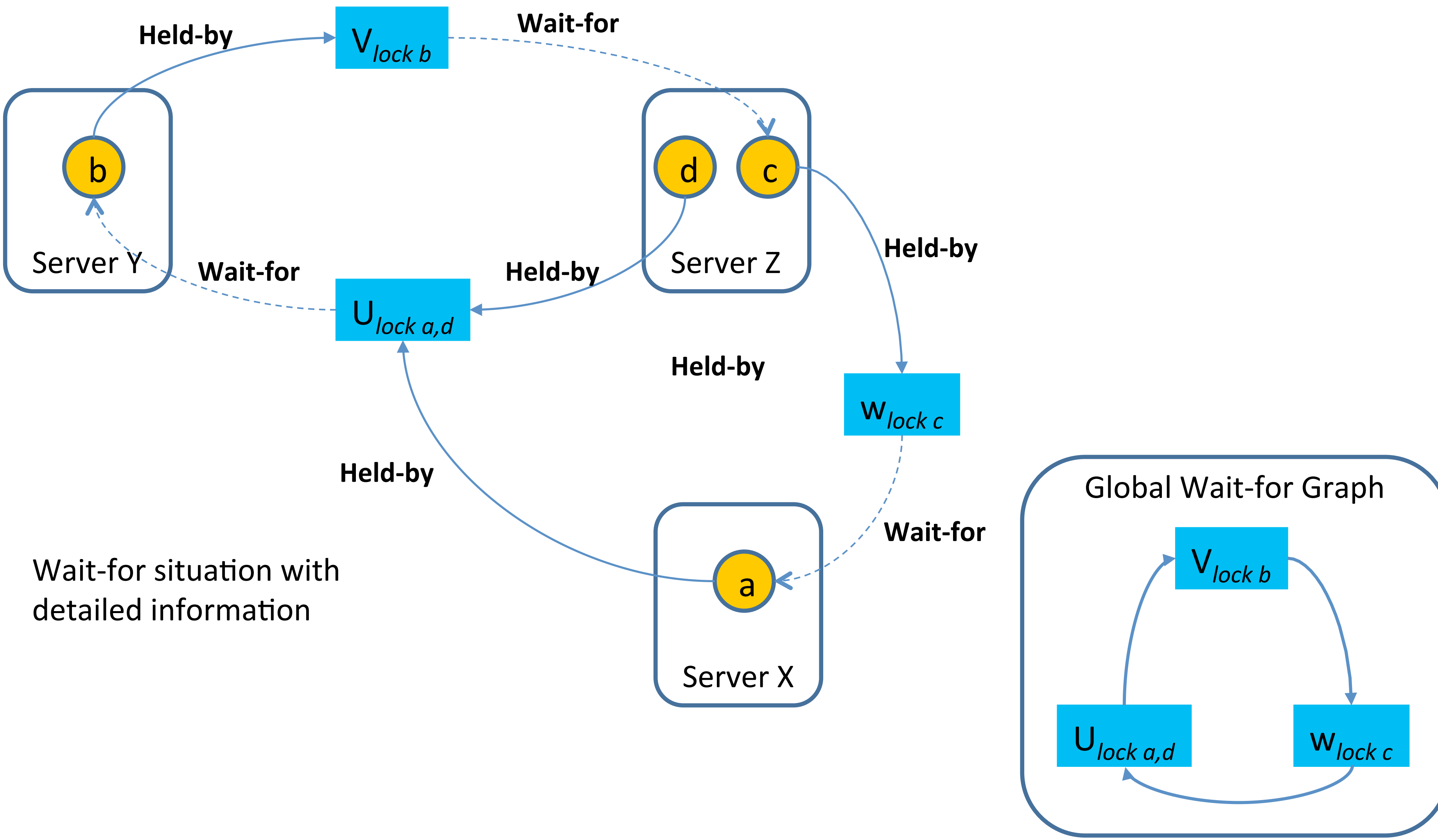
- Timeouts
  - Each lock is given a period of time where it is invulnerable
  - After this timeout, it becomes vulnerable
  - If transaction X holds a lock that becomes vulnerable and transaction Y is waiting for X, then X is aborted
  - Problem
    - Hard to decide on an appropriate length of timeout
    - Transactions may be aborted, when the lock becomes vulnerable and another transaction waits, but there is no deadlock
    - In overloaded systems, the number of transactions aborted increases
    - Long-lasting transactions may be penalized

Distributed Deadlocks

Distributed Deadlocks

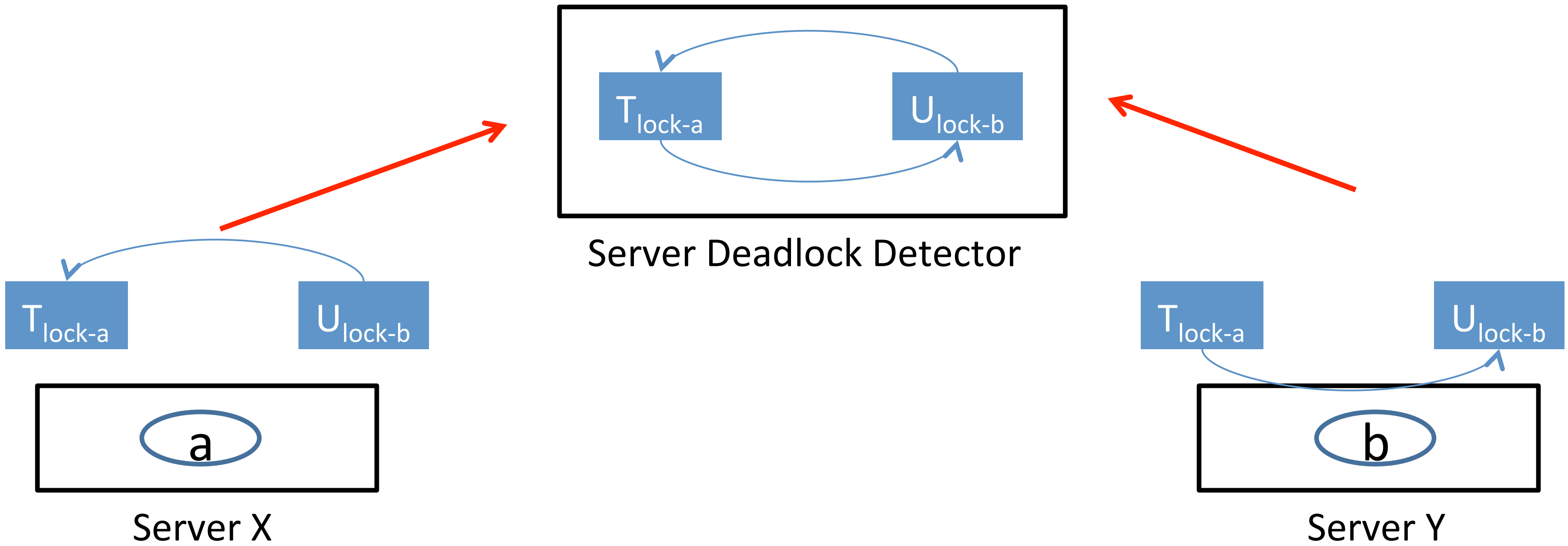
- Deadlocks are characterised by a loop in a wait-for graph
- In a distributed system, each server may utilise its own local wait-for graph for local deadlock detection
- However: there may be deadlock situations across servers
- How to detect global deadlocks across many servers?
  - Global wait-for graph: constructed from a combination of all local wait-for graphs
  - Please note: a loop may exist in the global graph that does not exist in any local graph

Global Wait-for Situation



Wait-for situation with detailed information

Centralized Detection



- Simplest solution – a single central server is responsible for detecting and breaking deadlocks

Centralised Detection

- Simplest solution – a single central server is responsible for detecting and breaking deadlocks
- All other servers periodically transmit their local wait-for graphs to the central server
  - It generates the global wait-for graph
  - It looks for loops
  - When a loop is detected, it makes a decision which transaction to abort to resolve the deadlock
  - It will instruct the servers responsible for these transactions to perform the required aborts

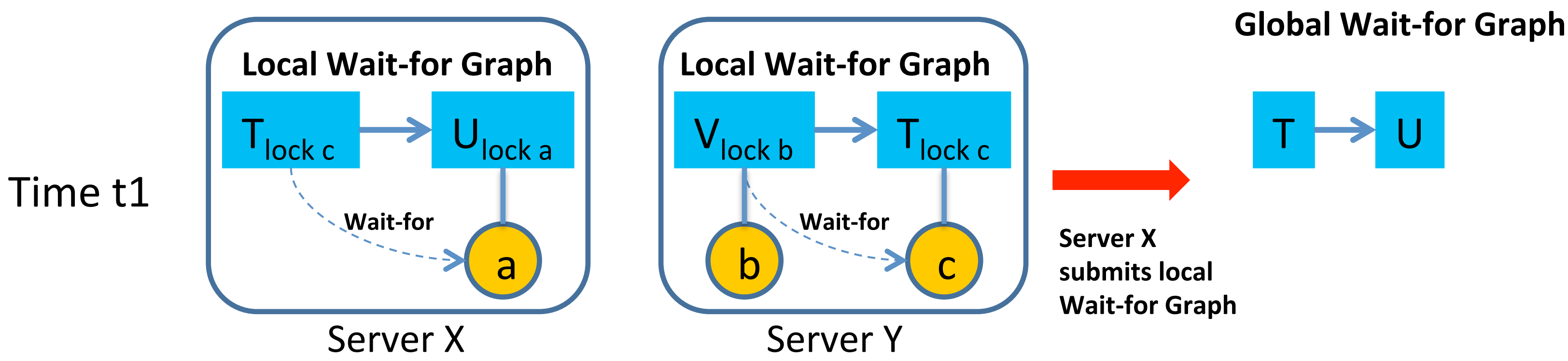
Centralised Detection: Problems

- Suffers from the usual problems of centralized solutions:
  - Single point of failure, lack of fault tolerance: if the central server crashes, no deadlock detection possible
  - Poor availability
  - Performance bottleneck: such a solution does not scale
- Communication overheads are high if local wait-for graphs are sent to the detector frequently
- Delays in detecting and resolving deadlock are high if local wait-for graphs are sent to the detector infrequently
  - May lead to wrong deadlock diagnosis – phantom deadlocks

Phantom Deadlocks

- Phantom deadlocks
  - A deadlock that is detected but not really a deadlock is called a "phantom deadlock"
- May occur due to time delays in transmitting information to a global deadlock detector
  - Global deadlock detector operates with outdated information
- E.g.: a transaction may already have released a lock locally, which has not been reported to global deadlock detector
  - the global wait-for graph is still constructed with this lock and may show a deadlock situation
- These "phantom deadlocks" may lead to unnecessary aborts

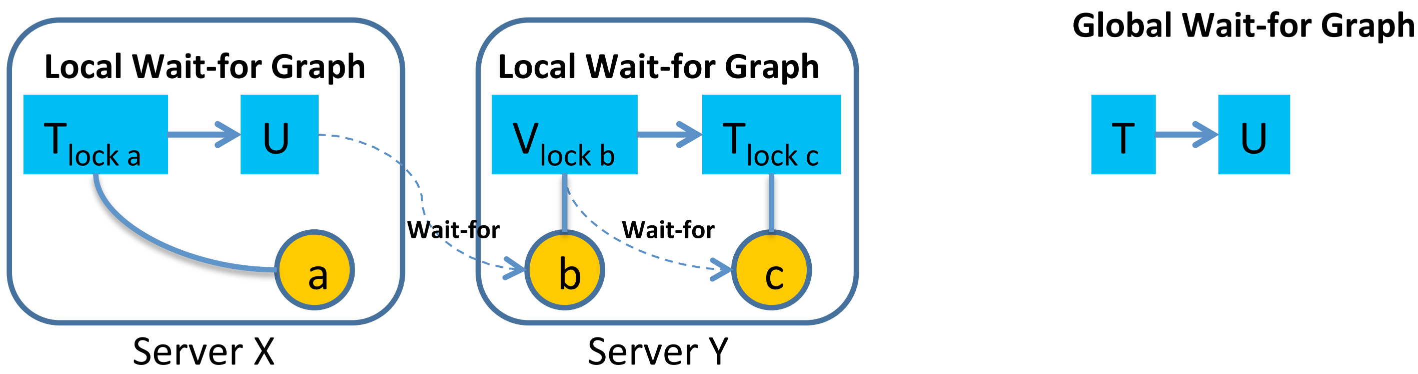
Phantom Deadlocks



- Situation: Transactions T, U, V lock objects on servers X and Y
- Wait situation at Server Y
  - Transaction T has locked object c
  - Transaction V has to wait for release of object c
- Wait situation at Server X
  - Transaction T has to wait for release of object a
- Server X reports local wait-for graph T --> U to Deadlock detector
- Server Y is delayed in its reporting: no knowledge about V --> T

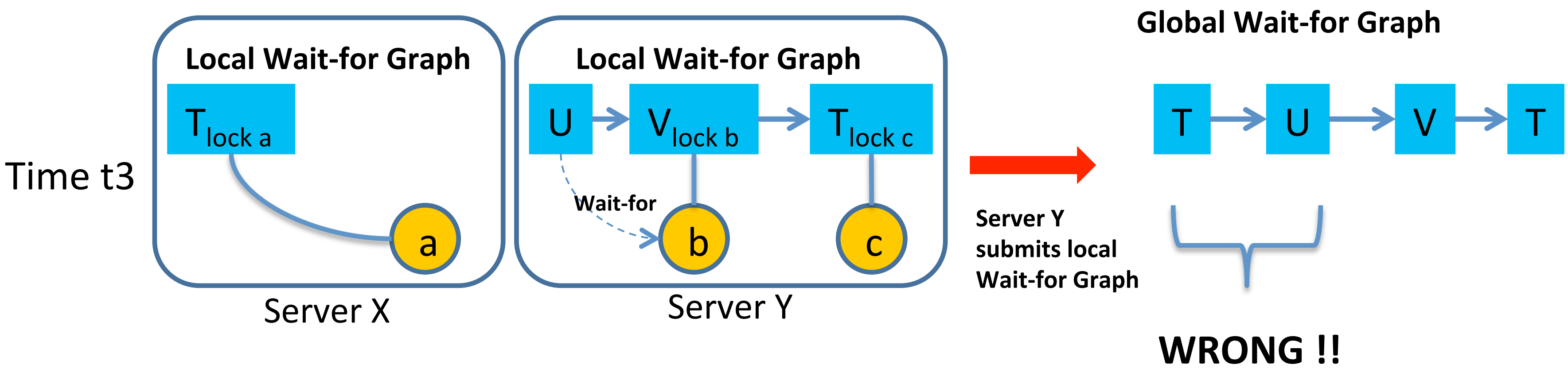
Phantom Deadlocks

Time t2



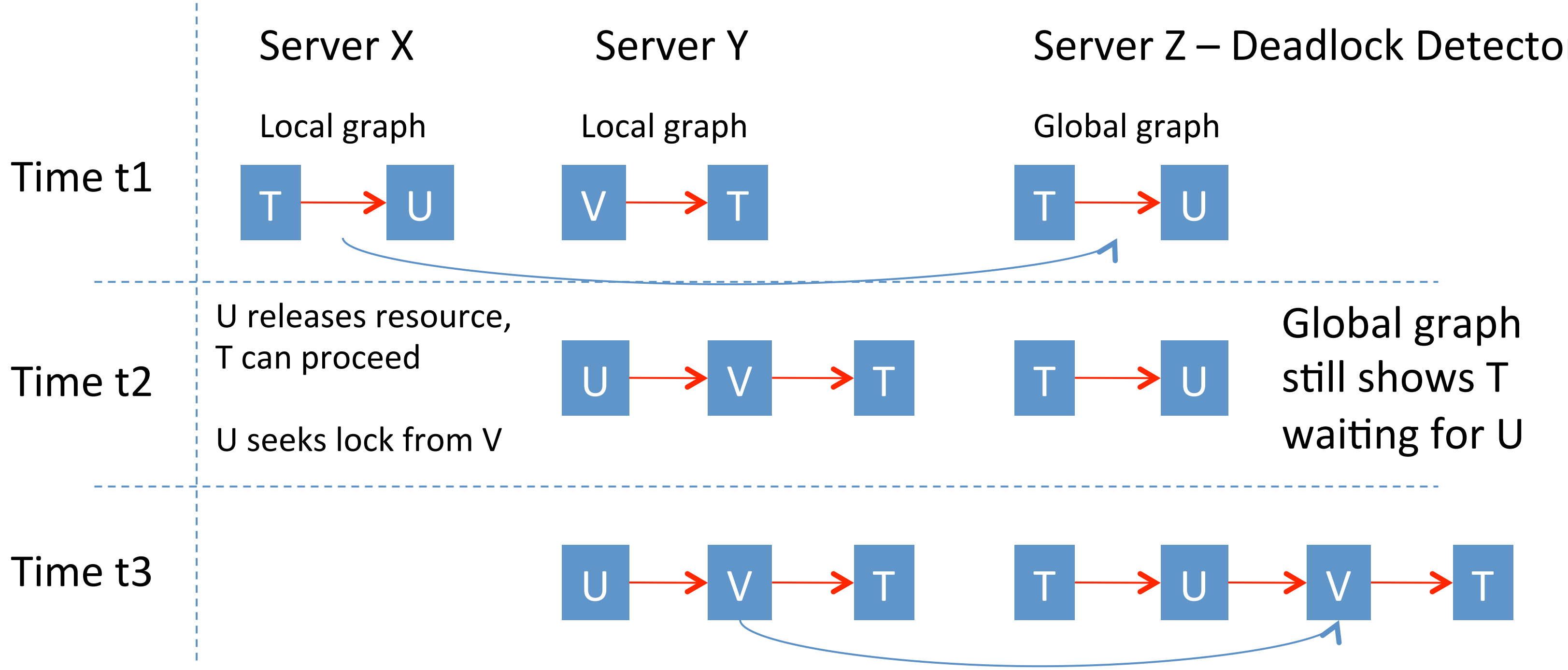
- Server Y is delayed in its reporting:
  - no knowledge about V --> T
- Transaction U releases object a and wants to lock object b on server Y
  - Transaction U starts waiting for transaction V to release object b
- **Global wait-for graph not updated !!**

Phantom Deadlocks



- Server Y updates local wait-for graph
- Server Y transmits local wait-for graph to global deadlock detector
- Deadlock detector constructs new global wait-for graph by merging the local graphs of servers X and Y
- However:
  - T is not waiting for U anymore !
- Global wait-for graph shows phantom deadlock !

Phantom Deadlocks



- At time t1, only local graph from server X is submitted – global graph only reflects the situation on server X
- At time t2, U releases resource, T can proceed
- At time t3, local wait-for graph of Server Y is submitted and merged into global wait-for graph – shows wrong situation: indicates a deadlock situation:
  - This will lead to an abort of transaction U or T, as the wait-for graph wrongly indicates that T waits for U

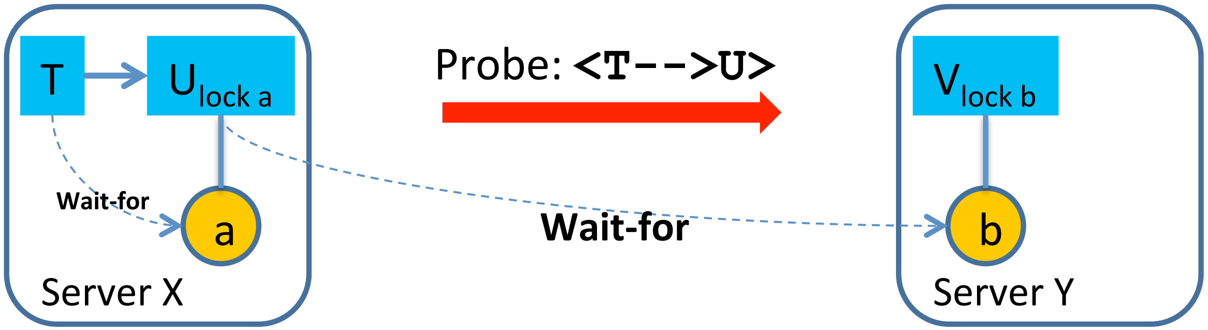
Edge Chasing

- Edge chasing is a distributed mechanism for detecting deadlocks
- It does not require the global wait-for graph to be constructed
- Servers find cycles using so-called “probes”
  – Probes are messages that are sent between servers and “follow” edges in a global “virtual” wait-for graph
- The probe collects information of the path taken through that graph
  – Records the wait-for situation of transactions
- A loop exists if a reference to a data object appears twice in the probe

Edge Chasing

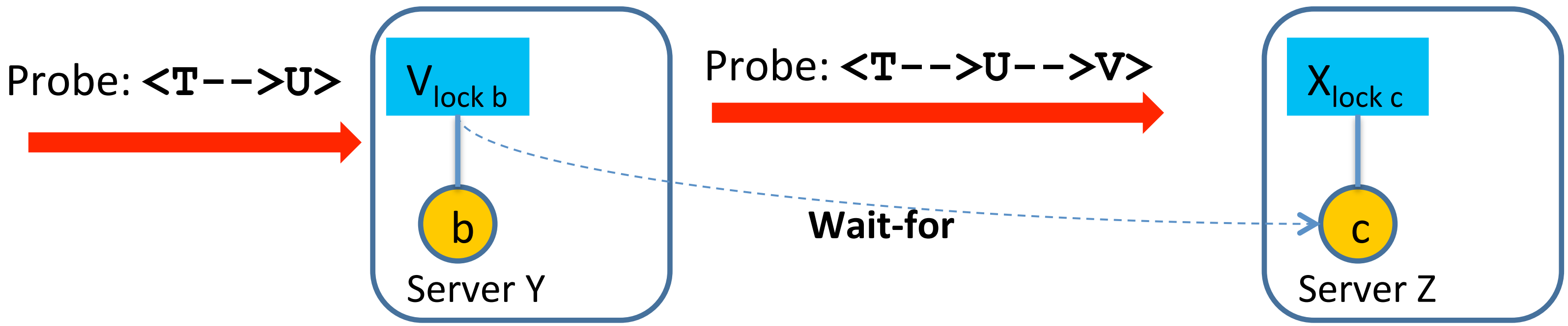
- Edge chasing has three steps:
  - Initiation: a probe is sent out
    - Whenever a transaction T is waiting for a resource to be released, a probe is sent to all transaction servers where the blocking transactions reside
  - Detection: a probe is received, deadlock detection, probe forwarding
    - If a probe is received by the server of a blocked transaction, the probe will be updated with this new transaction id and sent on to servers with the blocking transactions
    - If a probe is received at a server, where there are no blocked transactions then the probe is discarded
  - Resolution: breaking the deadlock

Edge Chasing: Initiating a Probe



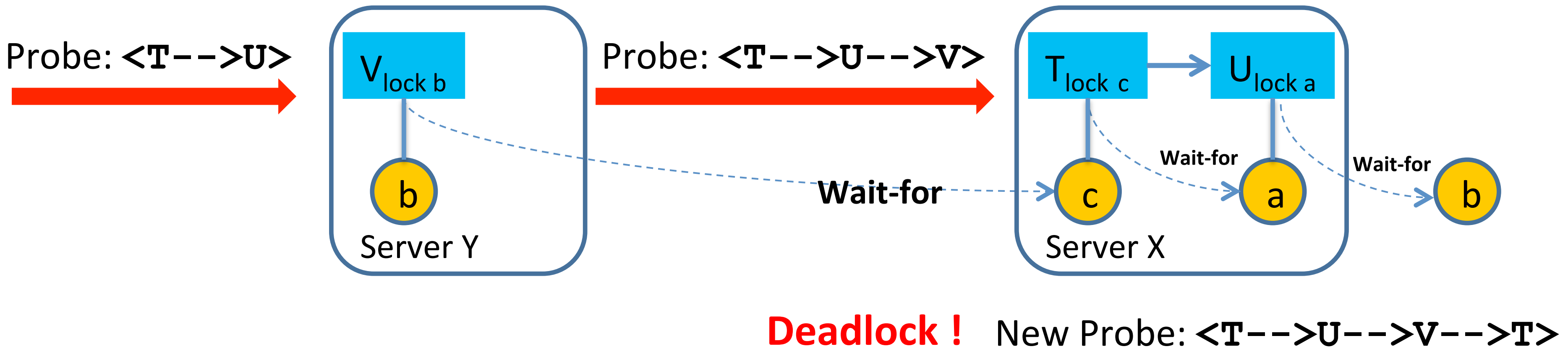
- A probe is initiated when
  1. A server notes that a transaction T starts waiting for another transaction U to release a resource
  2. Transaction U itself waits for release of a locked resource on another server
- The probe is initialised with the path <T-->U> and sent to the server where the locked resource is managed

Edge Chasing: Receiving a Probe



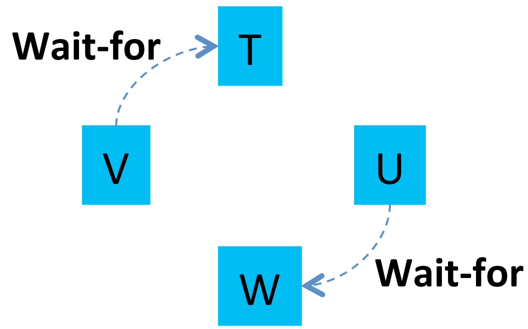
- When a server receives a probe <T-->U>, it first checks whether U is waiting itself for a resource to be released at the server:
  - If U is waiting for, e.g., transaction V to release b, then V is added to the path recorded by the probe, a new probe <T-->U-->V> is constructed
  - If transaction V itself, again, is waiting for an object on some other server, the probe is forwarded to that server

Edge Chasing: Detecting a Deadlock



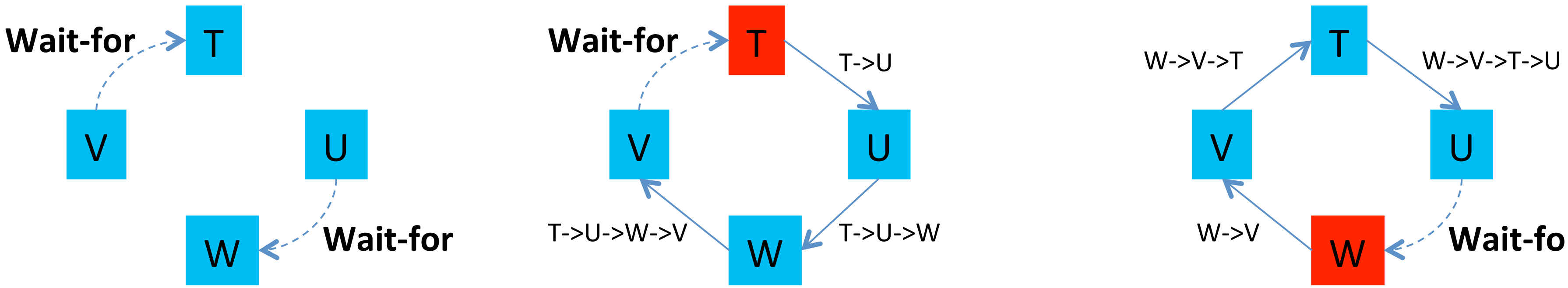
- When a server receives a probe, it also checks for a deadlock
  - If there is a transaction blocking access to a data object at the server and itself waiting, it is added to the probe
  - If the probe contains now a cycle – e.g. <T-->U-->V-->T>, then a deadlock has been detected
- One of T, U or V must be aborted

Prioritising Transactions



- Probes may be initialised by more than one server for the same wait-for loop
- A single deadlock will then be detected by two different servers
- What can happen: more than one transaction may be aborted to break a single deadlock loop!
  - We want to abort only as few transactions as possible

Prioritising Transactions



- Solution: place a total priority ordering over transactions (e.g. timestamp)
- Using priorities
  - When a deadlock is found, then the transaction with the lowest priority is aborted
  - Even if different servers detect the same deadlock cycle, the decision which transaction to abort will be the same

Limit the Number of Probes

- Can we use priorities to reduce the number of probe messages?
- Idea: probes only travel "downhill" – from transactions with higher priority to transactions with lower priority
  - This guarantees that deadlock detection is only initiated when a higher-priority transaction starts waiting for a low-priority transaction
- A probe is initiated when
  - A transaction, e.g. T, starts waiting for another transaction, e.g. U,
  - Transaction U is waiting for an object on some other server to become unlocked, and
  - Transaction T has a higher priority than U.
- However, this will not always detect deadlocks – the algorithm is not complete

Problem with Priorities

- If we assume that there are transactions U, V, W, with priorities that order them as U > V > W
- We assume that U waits for V and V waits for W
  - U --> V --> W
- Assumption: W starts waiting for U
  - Without priorities: a probe <W --> U> will be sent out to detect whether there is a deadlock and would eventually find it
  - With priorities: this probe is not sent out, as W has lower priority than U, deadlock is not detected

Queuing Probes

- We can make the algorithm complete by queuing probes
- Transaction coordinators store copies of all the probes received for a transaction in a probe queue
  - When a transaction, T, starts waiting for another transaction, U, the transaction coordinator of U saves the probe $\langle T \rightarrow U \rangle$ in its probe queue
  - When U starts waiting for V, the coordinator of U forwards its probe queue to the coordinator of V, which now has a probe queue with content $\langle T \rightarrow U \rangle$ and $\langle U \rightarrow V \rangle$
  - If V starts waiting for T, it will forward its probe queue to coordinator of T,
  - The coordinator of T recognises the dependency $\langle V \rightarrow T \rangle$ and combines it with the info in the received probe queue – deadlock detected
- However, it is now more difficult to guarantee correctness: coordinators must keep relevant probes and discard probes for completed transactions