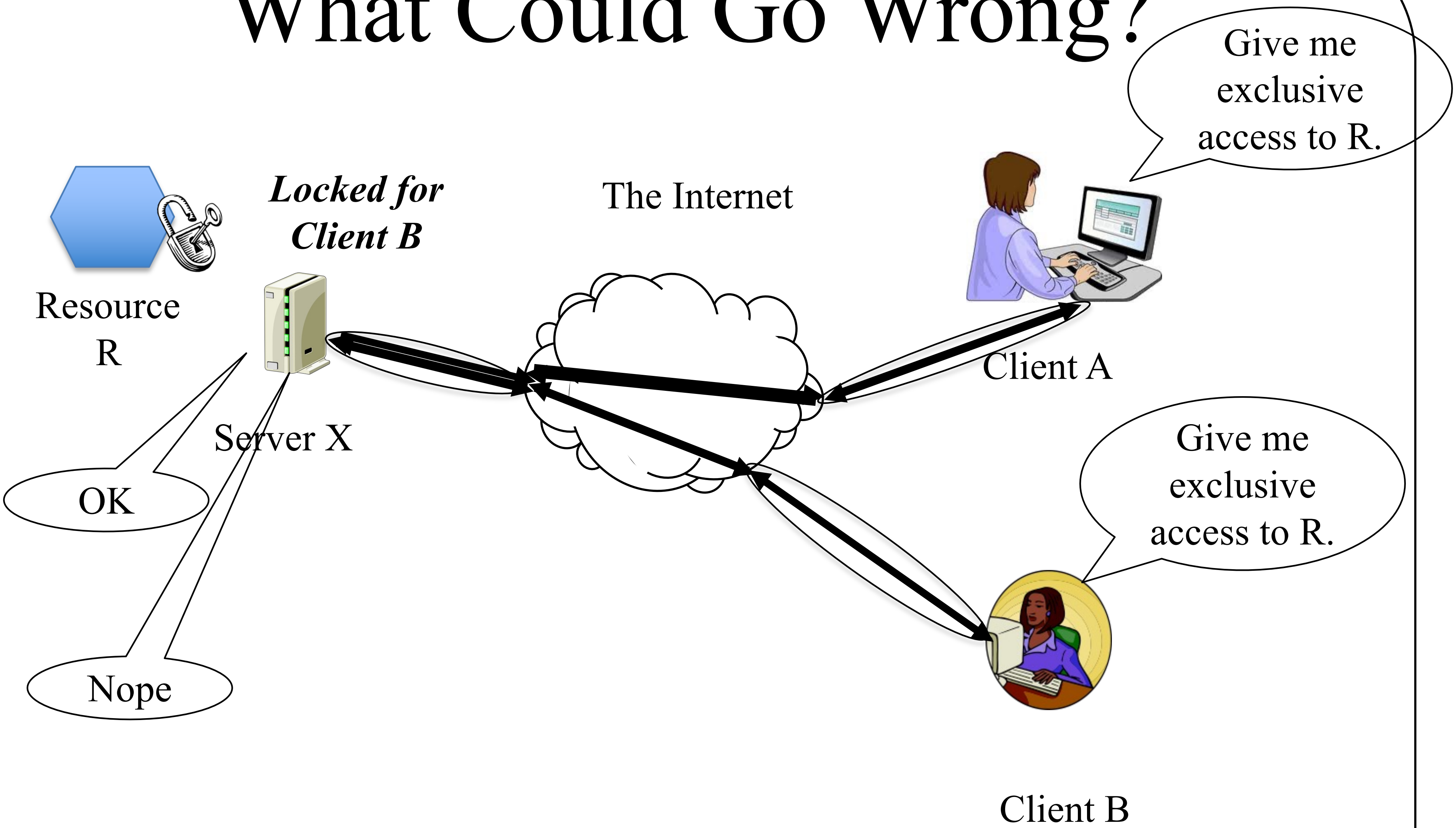
Distributed Systems: Synchronization and Consensus

CS 111
Summer 2025
Operating System Principles
Peter Reiher

Distributed Synchronization

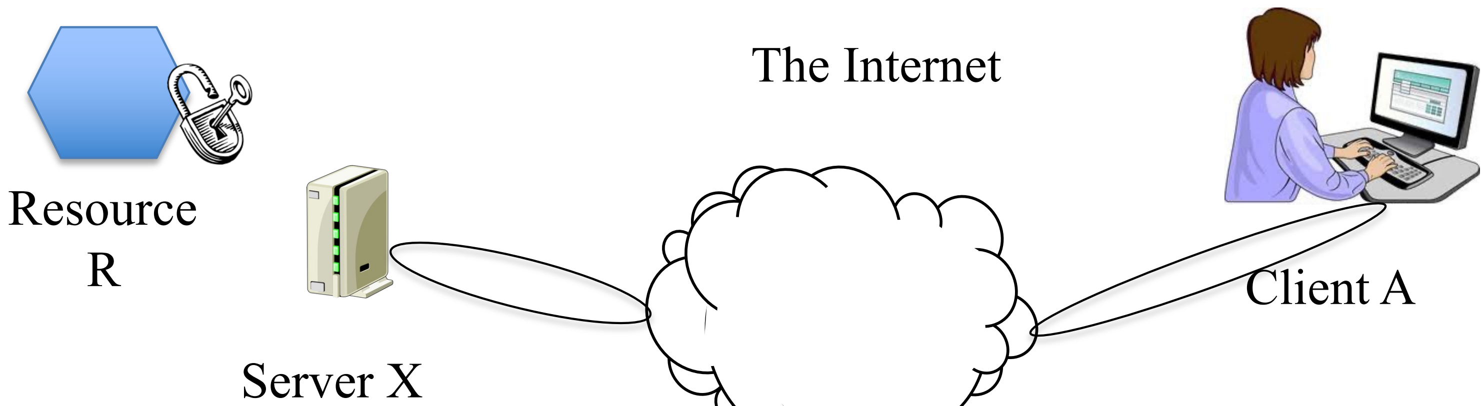
- Why is it hard to synchronize distributed systems?
- What tools do we use to synchronize them?

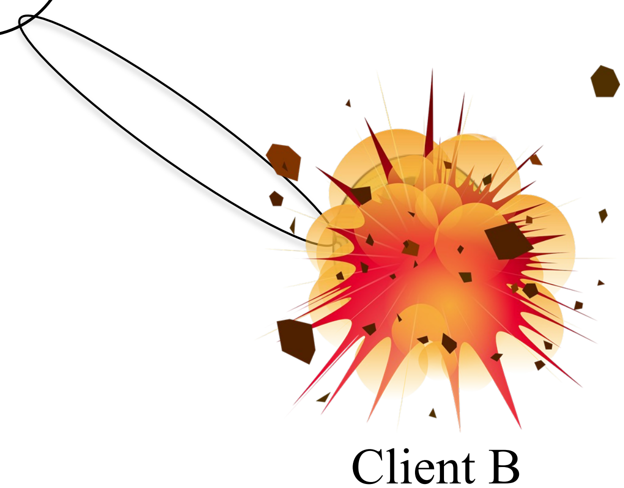
What Could Go Wrong?



Consider a simple case

One Possible Problem

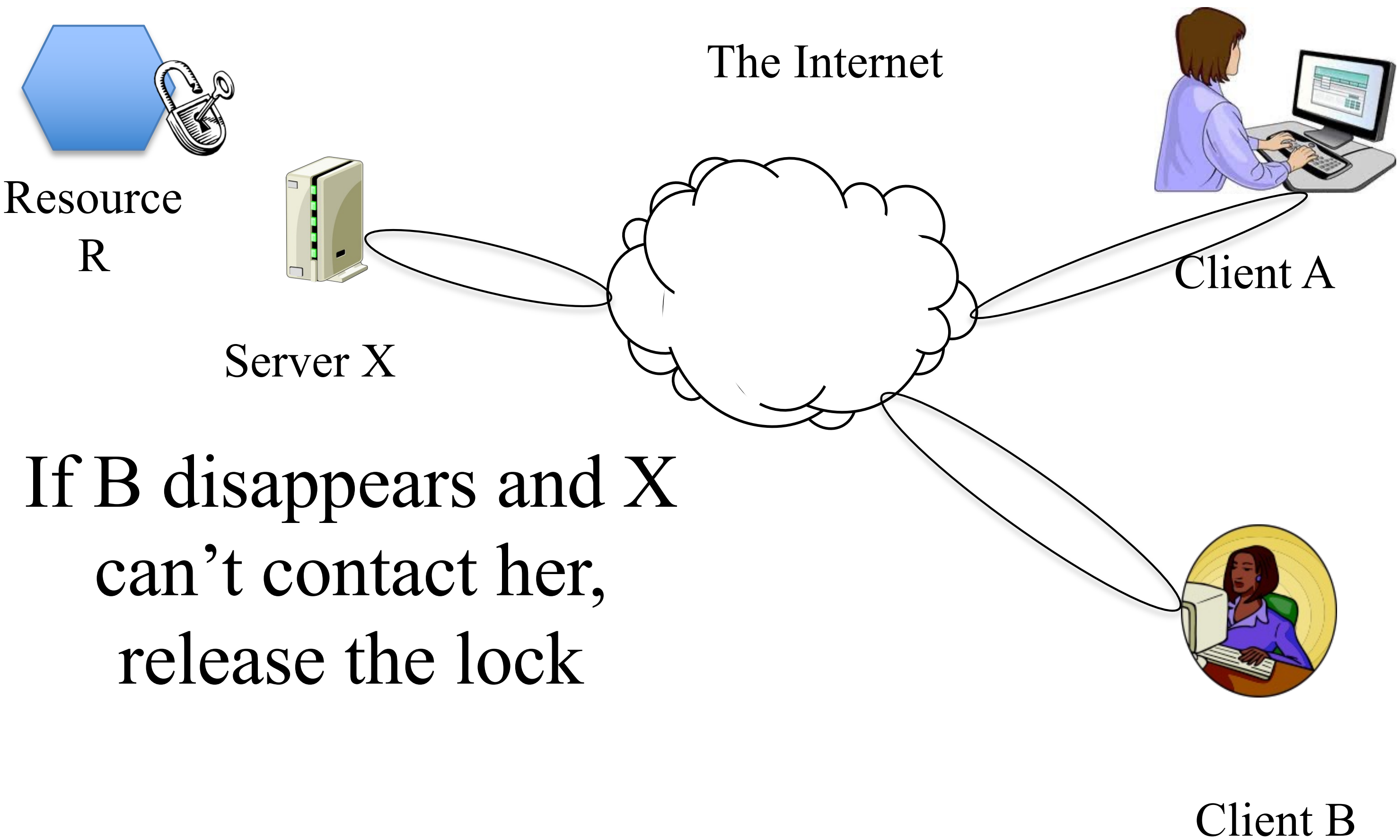




How can X know that the lock should be released?

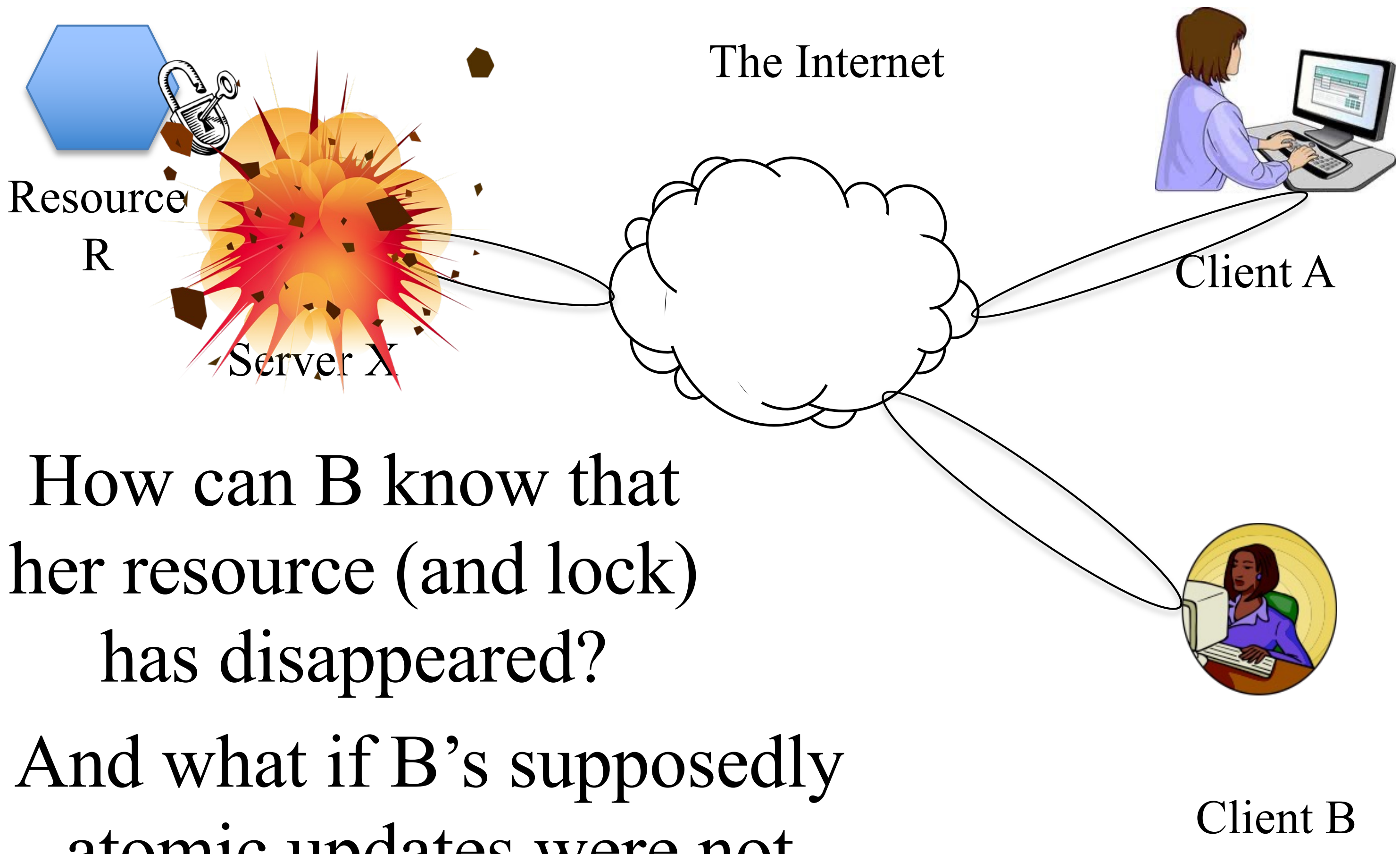
And what if B's supposedly atomic updates were not finished?

One Possible Answer



If B disappears and X can’t contact her, release the lock

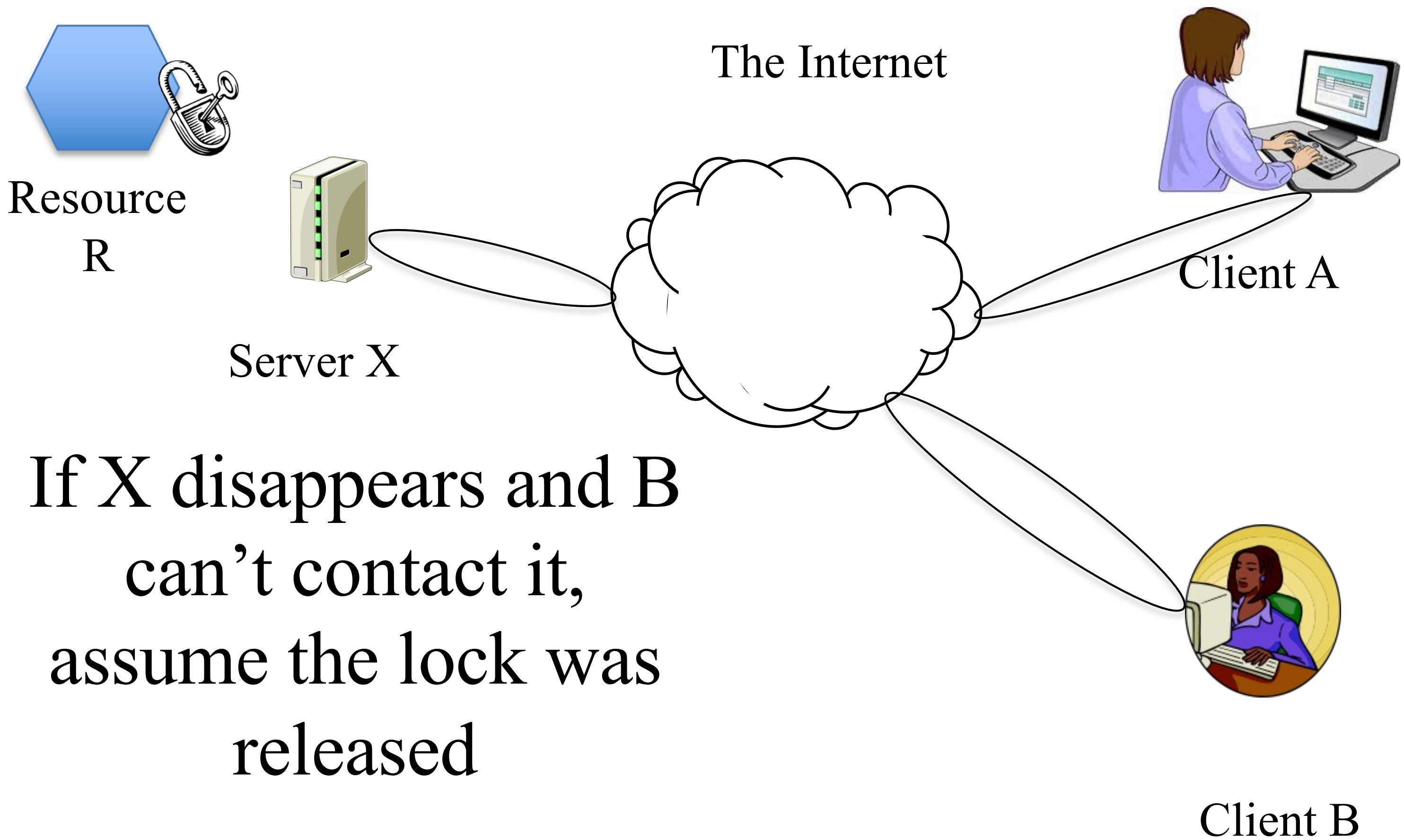
Another Possible Problem



How can B know that her resource (and lock) has disappeared?

And what if B's supposedly atomic updates were not finished?

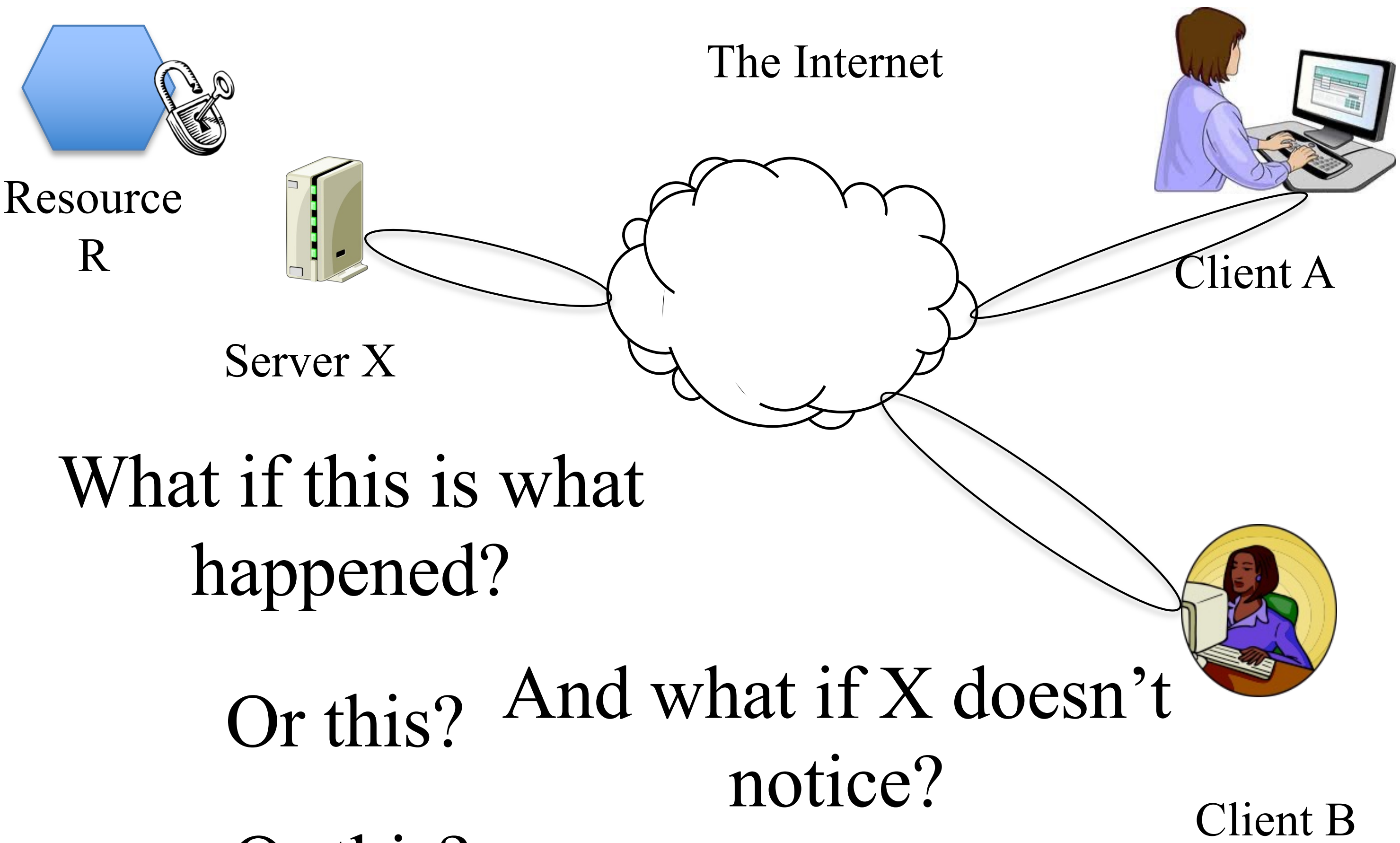
One Possible Answer



If X disappears and B can’t contact it, assume the lock was released

But . . .

And these are *simple* cases



What if this is what happened?

Or this?

And what if X doesn't notice?

Or this?

And what if the failed network comes back?

We'd Like Globally Coherent Views

- Everyone sees the same thing
- Everyone agrees if the lock is still present
- Everyone agrees if a machine has or hasn't failed
- Everyone agrees if a network is or isn't operating properly
- If a message is delivered, everyone sees the resulting state
- Usually the case on single machines

But . . .

- It's harder to achieve globally consistent views in distributed systems
  - Due to failures, recoveries, delays, etc.
- How to achieve it?
  - Have only one copy of things that need single view
    - Limits the benefits of the distributed system
    - And exaggerates some of their costs
  - Ensure multiple copies are consistent
    - Requiring complex and expensive consensus protocols
- Not much of a choice

What's Hard About Distributed Synchronization?

- Spatial separation
  - Different processes run on different systems
  - No shared memory for (atomic instruction) locks
  - They are controlled by different operating systems
- Temporal separation
  - Can't "totally order" spatially separated events
  - Before/simultaneous/after lose their meaning
- Independent modes of failure
  - One partner can die, while others continue

Locks in Distributed Systems

- Can we use locks to synchronize remote resources or computations?
- The situation:
  - Machine A wishes to obtain a lock for a resource X
  - The resource is on machine B
- So machine A probably needs to ask machine B to lock X
- And, if possible, machine B grants the lock and informs A

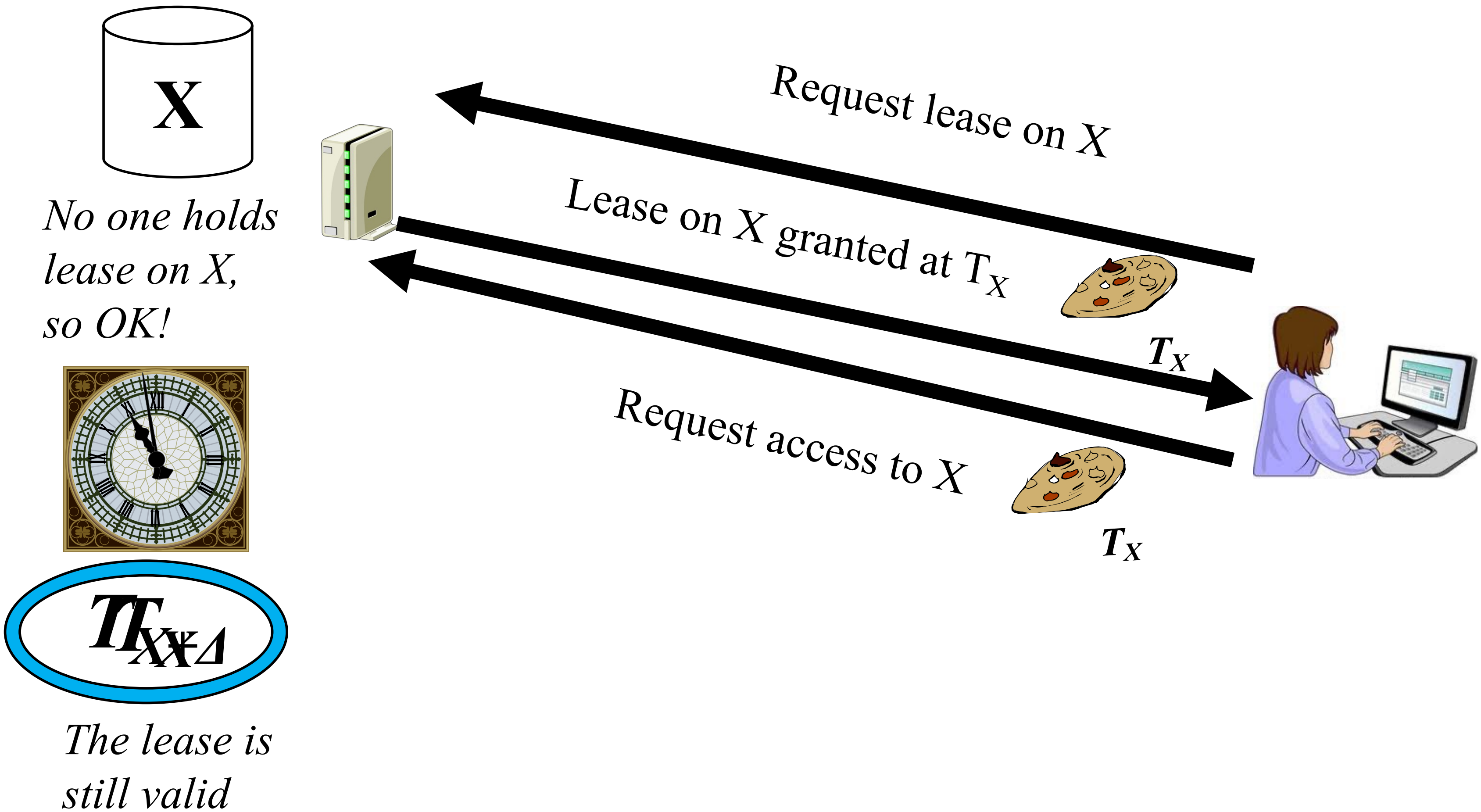
Problems With Locks in Distributed Systems

- So A has obtained a lock for X from B
- What if A fails?
  - In which case A won't release the lock
- What if A releases the lock, but the release message is lost?
  - In which case B doesn't know the lock should be released
- What if B fails?
  - When it recovers, will it remember that A locked the resource?
- Just some of the potential problems

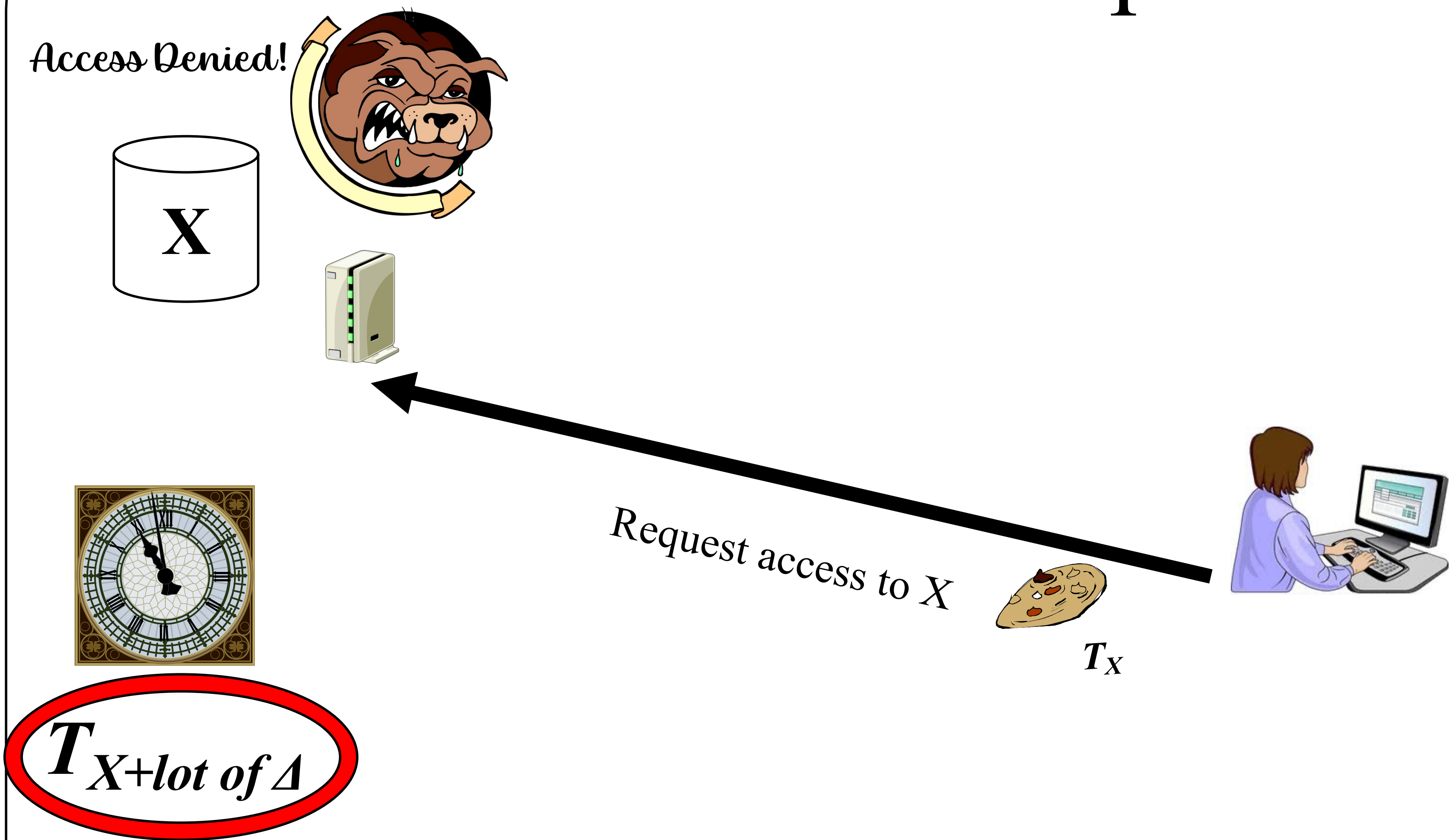
Leases – More Robust Locks

- Obtained from resource manager
  - Gives client exclusive right to update the file
  - Lease "cookie" must be passed to server on update
  - Lease can be released at end of critical section
- Only valid for a limited period of time
  - After which the lease cookie expires
    - Updates with stale cookies are not permitted
  - After which new leases can be granted
- Handles a wide range of failures
  - Process, client node, server node, network losses

Leases in Action



When the Lease Expires



The lease is no longer valid

Leases, Lock Breaking, and Recovery

- Revoking an expired lease is fairly easy
  – Lease cookie includes a "good until" time
    - Based on server's clock
  – Any operation involving a "stale cookie" fails
- This makes it safe to issue a new lease
  – Old lease-holder can no longer access object
  – But was object left in a "reasonable" state?
- Object must be restored to last "good" state
  – Roll back to state prior to the aborted lease
  – Implement all-or-none transactions

Complexity of Rollback

- Let’s say machine B has a lease on a resource from machine A
- Machine B does some work on the resource
  - Probably on a locally cached copy
- Machine B makes updates to local resources on the basis of that work
- The lease expires before machine B does all its operations

Now What?

- Some of the updates may have gotten to A
  - But not all of them
- If A has saved the state of the resource, it can restore it
  - Assuming A knows B wasn't finished
- What about B?
- B can discard all local changes to the resource
- But what about changes it made to other local resources?

It Gets Worse

- What if B also communicated to C based on its partial updates?
- And C did something on that basis
- B would need to remember that and tell C to undo its work
  - If C saved enough state to do so
- And, of course, C could have interacted with D . . .

Can We Ever Roll Back Everything?

- Under some circumstances, provably yes
- But you need to save lots of state about what happened
  - Both local updates and remote interactions
- The performance costs may be very high
- Very few systems even try
- At most, they restore the pre-update state of the leased resource, when necessary

Distributed Consensus

- Achieving simultaneous, unanimous agreement
  - Even in the presence of node & network failures
  - Required: agreement, termination, validity, integrity
  - Desired: bounded time
  - Provably impossible in fully general case
  - But can be done in useful special cases, or if some requirements are relaxed
- Consensus algorithms tend to be complex
  - And may take a long time to converge
- They tend to be used sparingly
  - E.g., use consensus to elect a leader
  - Who makes all subsequent decisions by fiat

Typical Consensus Algorithm

1. Each interested member broadcasts his nomination.
2. All parties evaluate the received proposals according to a fixed and well known rule.
3. After allowing a reasonable time for proposals, each voter acknowledges the best proposal it has seen.
4. If a proposal has a majority of the votes, the proposing member broadcasts a claim that the question has been resolved.
5. Each party that agrees with the winner's claim acknowledges the announced resolution.

6. Election is over when a quorum acknowledges the result.

What's going to happen if someone lies . . . ?

One Sample Leader Election Algorithm

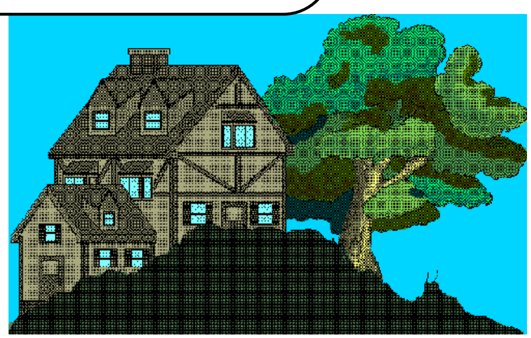
- The bully algorithm
- Choose a leader and follow its instructions
- Who gets to be the leader?
- Consider a group of children choosing a leader
- The biggest kid on the block gets to be the leader
- But what if the biggest kid on the block is taking his piano lesson?
- The next biggest kid gets to be leader
  – Until the piano lesson is over . . .

Electing a Bully

Spike's piano lesson
The Mom isn't let him out yet

The kids come out to play

I'm the leader, and we're playing baseball!
I'm here, where are you guys?
Hey, Butch!
I'm the leader, let's play tag!
Hey, Spike!
Hey, Cuthbert!



Assumptions of the Bully Algorithm

- A static set of possible participants
  - With an agreed-upon order
- All messages are delivered with T_m seconds
- All responses are sent within T_p seconds of delivery
- These last two imply *synchronous behavior*
  - A guarantee that things happen within a maximum period of time

The Basic Idea Behind the Bully Algorithm

- Possible leaders try to take over
- If they detect a better leader, they agree to its leadership
- Keep track of state information about whether you are electing a leader
- Only do real work when you agree on a leader

The Bully Algorithm and Timeouts

- Call out the biggest kid’s name
  - If he doesn’t answer soon enough, call out the next biggest kid’s name
  - Until you hear an answer
  - Or the caller is the biggest kid
  - Then take over, by telling everyone else you’re the leader

The Bully Algorithm At Work

- In a distributed system, rather than children in a neighborhood
- One node is currently the coordinator
- It expects a certain set of nodes to be up and participating
- The coordinator asks <u>all</u> other nodes
- If an expected node doesn't answer, start an election
  - Also if it answers in the negative
- If an unexpected node answers, start an election

The Practicality of the Bully Algorithm

- The bully algorithm works reasonably well if the timeouts are effective
  - A timeout occurring really means the site in question is down
- And if messages are not often lost or delayed too much
- And there are no *partitions* at all

Partitions and Distributed Systems

- Let's say there are *n* participating nodes
- What if nodes 1 through *n/2* can communicate with each other
  - Call them group 1
- And nodes *n/2*+1 through *n* can communicate with each other
  - Call them group 2
- But no one in group 1 can communicate with anyone in group 2
  - And vice versa

The Effects of Partitions

- Various members of the system have different views of the world
- Leading to different behaviors in each part of the system
- Even more complicated if there are more than 2 partitions
- And even more so if partitions split, merge, and re-split in complex ways
- Particularly bad if partitions split and merge frequently

How To Deal With Partitions?

- Try real hard to avoid them, mostly
- Less likely on LANs
- Or if there are redundant network paths between all participants
- When possible, design networks with these characteristics
  - E.g., in a cloud environment
- Often system designers ignore the possibility

Conclusion

- Distributed systems face serious challenges in providing even simple synchronization
  - Leases can avoid some of the downsides of locks (at a cost)
- Reaching any kind of agreement in distributed systems can be challenging
  - Generally best to elect a leader
- Partitions are a danger that's often hard to avoid
  - But are worse to deal with