Prob 1a

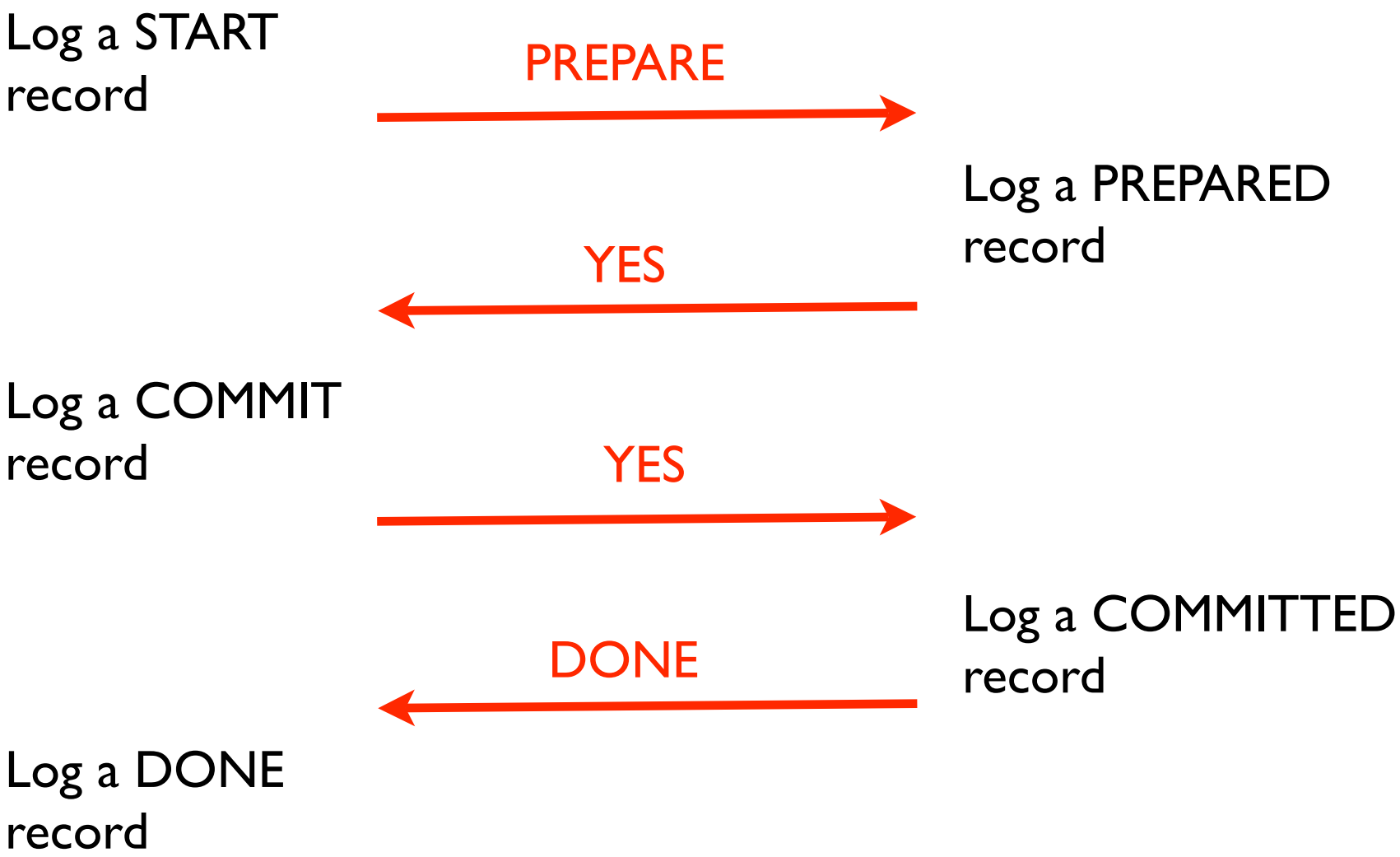
Problem 1: 2PC (20 pts)

Consider a distributed TP system comprising a commit coordinator and `n` resource managers (participants). Let

- `RTT[i]` be the expected rount-trip time for a message exchange between the coordinator and participant `i`;
- `WT[i]` be the expected time to write a log record at participant `i`;
- `WT[0]` be the expected time to write a log record at the coordinator;
- `PT[i]` be the expected time for participant `i` to attempt to prepare and determine its vote;
- `CT[i]` be the expected time for participant `i` to commit from the prepared state.

(a) What is the expected latency of a successful 2PC involving all `n` participants?

Logging in 2PC



Prob 1a, b

(a) What is the expected latency of a successful 2PC involving all n participants?

WT[0] -- coord logs START record
+ max { RTT[i] + WT[i] + PT[i] | 1<=i<=n } -- participants vote
+ WT[0] -- coord logs COMMIT record
+ max { RTT[j] + WT[j] + CT[j] | 1<=j<=n } -- participants commit

This makes the (questionable) assumption that the time to send a message is small compared to its RTT. Also we do not count the time the coordinator is writing the DONE record as part of the 2PC.

(b) Why doesn't this imply an upper bound on the throughput?

Because the coordinator and participants are free to execute as many txns in parallel as they wish.

Group commit techniques will slightly increase latency of individual txns but greatly improve throughput.

Prob 1c

(c) What is the expected number of txns in 2PC as a function of TPS?

Suppose on average there are N txns in 2PC and each 2PC requires T seconds to complete. Then

$TPS = N/T \qquad N = TPS * T$

and by specializing the answer to part (a) we get

$T = (2WT[0] + 2RTT + 2WT + PT + CT)$

$N = TPS * (2WT[0] + 2RTT + 2WT + PT + CT)$

Prob 1d

(d) What is the expected number of in-doubt txns resulting from a coordinator failure?

A txn is in-doubt at a participant if the participant has voted “YES” and not received a COMMIT or ABORT message from the coordinator.

If coordinator has sent the PREPARE, we assume the participant will eventually vote YES, and be in-doubt unless the coordinator sends the COMMIT message.

N * (RTT + PT + WT + WT[0]) / T

Prob 1e

(e) What bad effect results from locating the participants far from the coordinator?

This increases RTT.
From the previous parts, both N and T grow linearly with RTT.
The number of in-doubt txns generated by a coordinator crash grow as

N * (... RTT ...) / T

which is asymptotically

THETA(RTT)

Thus the in-doubt txns will increase in proportion with the RTT.

Prob 2

Consider a business transaction that does multiple transfers; for example

subtract $10 from account A
add $10 to account B
subtract $5 from account C
add $5 to accound D
subtract 1 widget from warehouse
add 1 widget to truck

There are *normal* reasons to abort (such as finding no widgets left in the warehouse) and *exceptional* ones (such as an application crashing). Ideally, we would like to implement the business transaction so all the above actions happen (in some order) as separate ACID transactions, and we are able to roll back in response to *any* (normal or exceptional) condition.

Is this possible? If so, show how. Otherwise, do as well as you can, describe the situations under which you cannot roll back cleanly, and explain the problem.

Prob 2

- Degrees of freedom:
  - Reorder
  - Introduce ACID txns combining individual actions
  - Split actions to separate out parts that cannot be compensated
  - Distinguish normal vs exceptional failures
  - Parameterize rollback by failure class

Prob 2

For discussion:

1. subtract one widget from warehouse
   (move it to loading dock)
2. subtract $10 from acct A
3. subtract $5 from acct C
4. add one widget to truck
   (move it from loading dock)
5. add $10 to acct B
6. add $5 to acct D

But you may not be allowed to withold payment for that long!

Prob 2

A second solution:

1. subtract one widget from warehouse inventory
2. subtract $10 from acct A
3. subtract $5 from acct C

-- all the above happen before user logs off!

4. move one widget from warehouse to loading dock
5. load widget on delivery truck
6. verify successful delivery
7. add $10 to acct B
8. add $5 to acct D

Prob 3

Suppose your database server has a capacity of at most `cr` read accesses per second, or a smaller number `cw` of write accesses per second, or any convex combination

```
a * cw  +  (1-a) * cr
```

of read and write accesses.

Despite the current state of the economy, your business is growing, and the offered read and write load on your database (`lr` and `lw`, respectively) has reached its capacity. So you consider the following schemes for expanding to two (or in general to `n`) servers:

Replication: Replicate all your data on all `n` servers. Implement each write using 2PC to write every copy of the datum in the same distributed ACID transaction. (Your friendly sales rep assures you that the 2PC is no more expensive than the sum of the write transactions at each of the `n` servers.) Implement each read by choosing a server at random or by some more sophisticated load balancing scheme.

Partitioning: Assign each row of data to one server according to some hash function applied to its primary key: if `hash(k) = i` then the row with key value `k` is stored on database `i`.

Prob 3a

(a) What is the capacity of an n-server system using each of these schemes, assuming all reads and writes are indexed by primary key? There is plenty of main memory available, so you may make the (only slightly unrealistic) assumption that the PK index is entirely cached in main memory at each server, and the individual server capacities are independent of the amount of data stored at them.

Note the simple "convex combination" description used above to describe a single server doesn't work for the partitioning scheme without some assumptions about the distribution of write keys, and it doesn't work at all for the replication scheme. The answer is a bit more subtle than that.

Discuss the scalability of the two schemes.

Partitioning (assume uniform distribution of keys):

$cw[n] = n * cw[1]$ $cr[n,w] = n * cr[1,w/n]$

Replication:

$cw[n] = cw[1]$ $cr[n,w] = n * cr[1,w]$

Partitioning scales better but is not fault-tolerant.

Prob 3b

(b) Suppose the offered read load includes some reads using a secondary key. An individual database server is indexed on both keys, and its capacity is described by a triple `cw`, `crp` and `crs`, reflecting writes, reads by primary key, and reads by secondary key. How does the answer to part (a) change?

Partitioning:

cw[n] = n * cw[1] crp[n,w] = n * crp[1,w/n]
crs[n,w,rp] = crs[1,w/n,rp/n]

Replication:

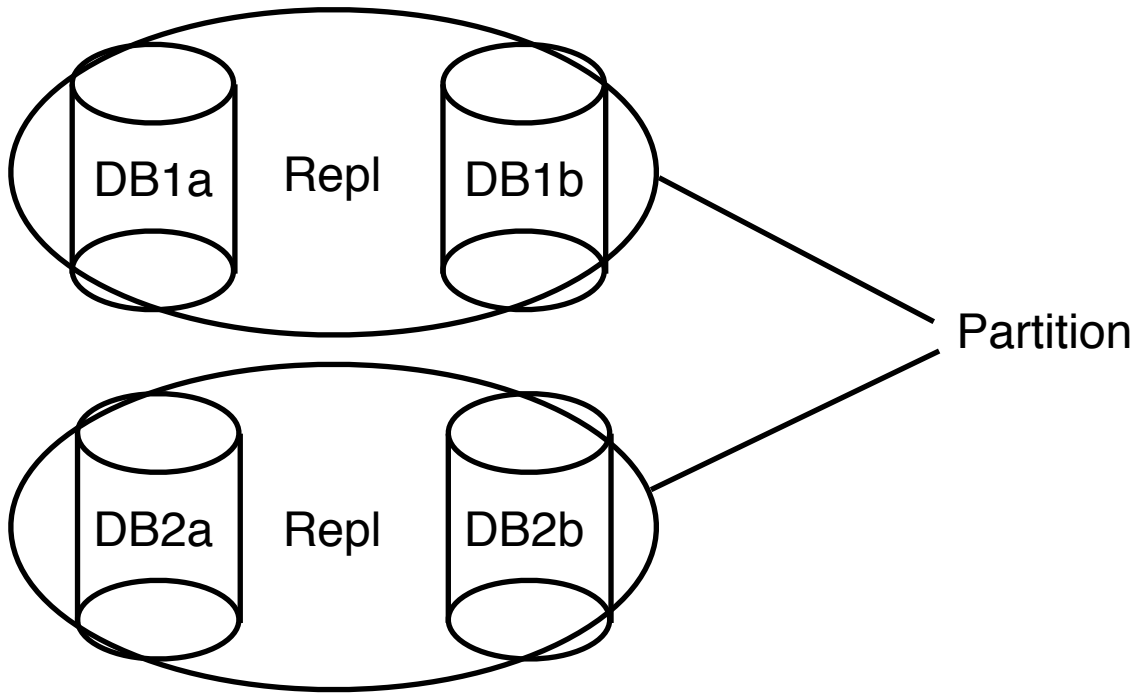
cw[n] = cw[1] crp[n,w] = n * crp[1,w]
crs[n,w,rp] = n * crs[1,w,rp/n]

(It changes a lot!)

Prob 3c

(c) In the real world, failures happen. Obviously, the partitioning scheme described above cannot cope with a database failure.

Describe a scheme that combines replication and partitioning to remedy this, while retaining some of the advantages of the pure partitioning scheme.



Prob 3c

Suppose there is an *m*-way partition of *k*-way replicated clusters.

cw[m,k] = m * cw[1,k]
cw[1,k] = cw[1,1]

crp[m,k,w] = m * crp[1,k,w/m]
crp[1,k,w] = k * crp[1,1,w]

crs[m,k,w,rp] = crs[1,k,w/m,rp/m]
crs[1,k,w,rp] = k * crs[1,1,w,rp]

Prob 4

In Lecture 6 we discussed (forward) recovery for multi-transaction RPC when some actions might not be recoverable. There are two basic approaches: idempotency, and readable device state that reliably reflects the operations the device has performed.

We gave an example with a simple device that performed exactly one operation per transaction. Now consider a more complicated example: a (high-end) ATM that

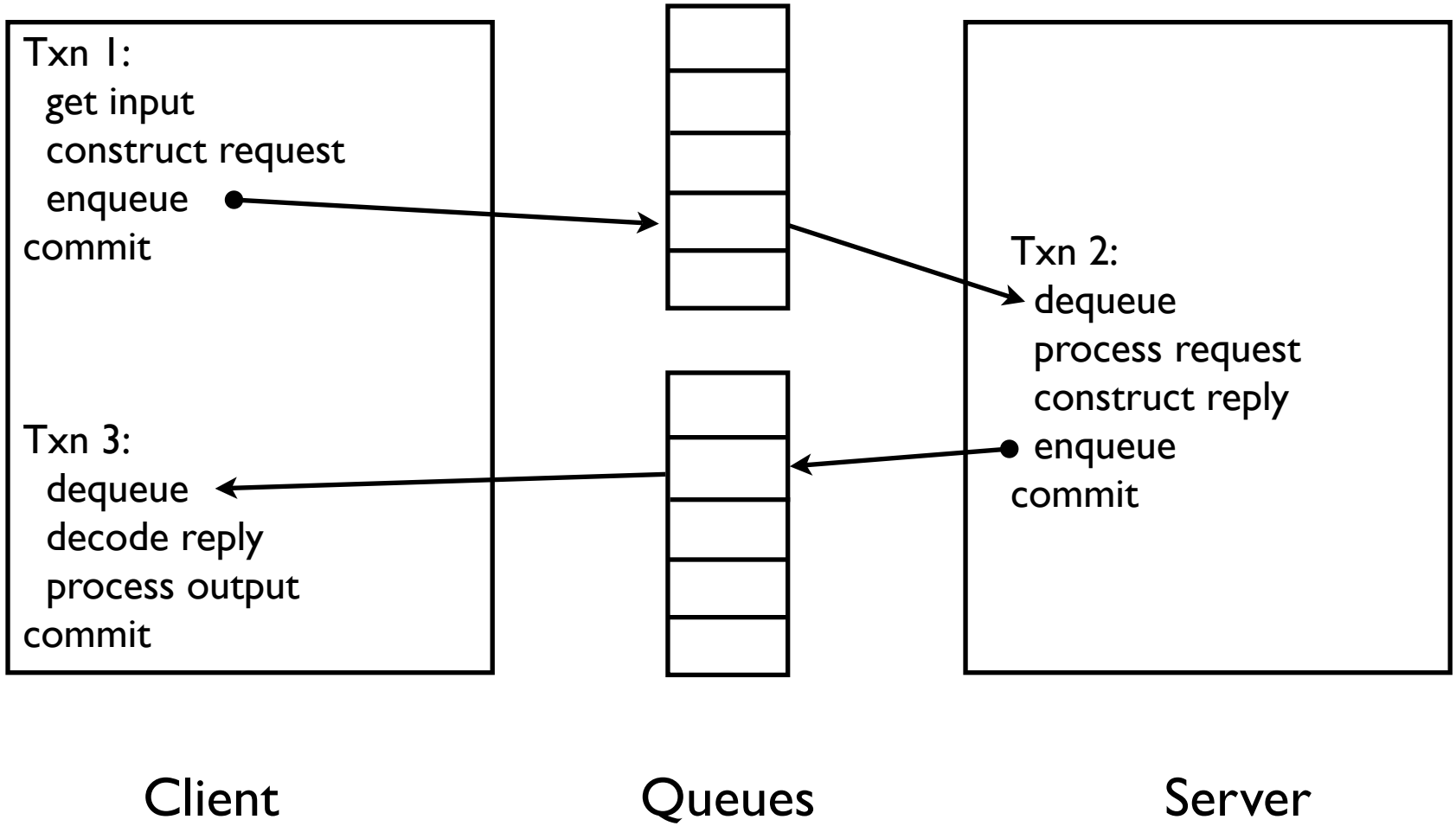
(1) gives out some paper money;
(2) gives out some change; and
(3) prints a receipt.

Can you extend our recovery example so it works properly with a device like this that makes multiple state transitions as part of the commit?

If so, describe how to do it, including the assumptions you make about the behavior of the device state.

If not, explain why.

Multi-Transaction RPC



Idea: suppose client or server fails; on recovery, can we recover the state of current RPC?

Client Recovery

Assume: client runs one txn at a time.

On recovery: 4 possible states of current txn:

A: Txn 1 did not commit
Client should resubmit request if possible

B: Txn 2 did not commit
Client should wait

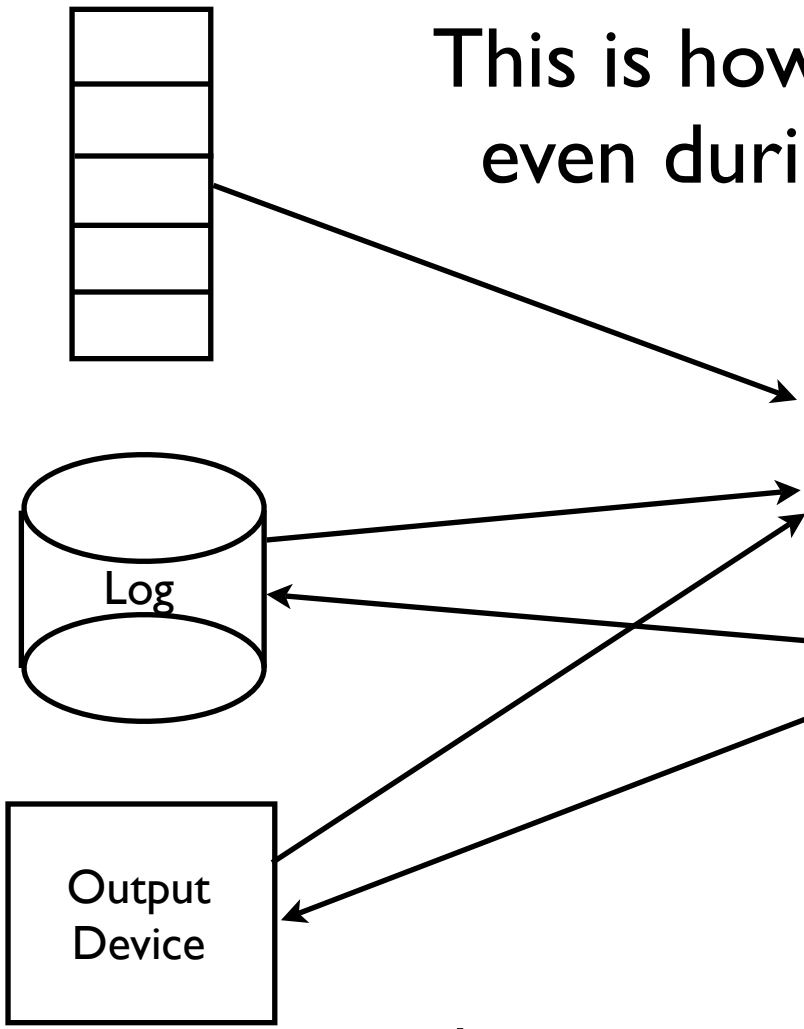
C: Txn 3 did not commit
Client should restart txn 3

D: Txn 3 committed
No client action required

Recovering from state C

- We don't know how far client got in previous run ...
- Might have done something that cannot be undone/redone
  - print boarding pass
  - dispense cash
- How do we handle this? Two ways:
  - device state
  - idempotency

Recovering Using Device State



This is how to run T3 so it works properly even during recovery ...

1. start transaction T3
2. dequeue reply
3. earlier logged device state different from current state => ask operator
4. write current device state to log
5. perform operation on device
6. commit

Assumption: every operation changes state of device!

Prob 4

Assumption: reliable device state -- device either

- succeeds, changes state in predictable way, or
- fails, changes to error state, and
- no state is ever repeated

<txn-num, step-num>

1. start transaction T3
2. dequeue reply
3. earlier logged device state txn-num different from current => ask operator
4. device in error state => ask operator
4. write current txn-num to log
5. perform operations on device starting with device current step-num
6. commit

Prob 4

Note:

Printing the receipt can be made idempotent simply by including the transaction number on the receipt.

The effect of printing the receipt twice is very like that of a Xerox machine ...

Prob 5

Most eCommerce sites use cookies to help identify sessions, (though storing the entire session state in a cookie is rare). In lecture we discussed a fault-tolerance scheme using reliable multicast (e.g. the Spread toolkit) to maintain synchronized replicated copies of session state in all the application servers.

With the same session state available to every application server, it might seem that the load balancer could ignore session information in connection requests, e.g. routing each new connection request to the currently most-lightly-loaded server. This might allow the load balancer to avoid switching by layers 4-7, thus avoiding the expensive "delayed binding" required in order to examing higher-level protocol data.

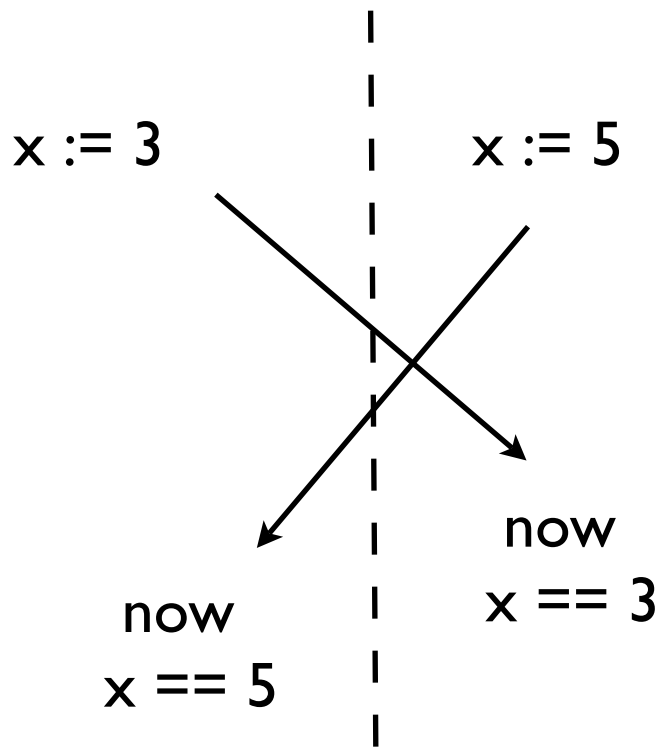
Unfortunately, with many browser clients it is possible to send two or more http requests (in the same cookie environment, therefore the same session) without waiting for a response from the server.

What problems could this create? Suggest one or more possible efficient solutions.

Prob 5

The problem:

Processes in different App Server instances could end up working on the same session concurrently leading to inconsistent states at the two servers ...



Prob 5

The solution: There is no *really good* solution.

1. Keep session state in cookie is *not correct*.
   (cookies for concurrent executions are
   inconsistent, one overwrites another at client)

2. Delayed binding and sticky sessions: *expensive*.

3. Keep session state in database: *expensive*.

4. Build specialized lock manager using shared memory
or a fast interconnect: *costly* and *complex*.
   (like re-implementing database lock mgr or Spread)

5. Write the application so concurrency without locks
does not matter: *very hard*.