95-702 Distributed Systems Transactions and Concurrency Control

Transaction Notes mainly from Coulouris

Distributed Transactions Notes adapted from Tanenbaum's "Distributed Systems Principles and Paradigms"

Transactions

- A transaction is specified by a client as a set of operations on objects to be performed as an indivisible unit by the servers managing those objects.
- The servers must guarantee that either the entire transaction is carried out and the results recorded in permanent storage or, in the case that one or more of them crashes, its effects are completely erased.

Transactions (ACID)

- **Atomic**: All or nothing. No intermediate states are visible.
- **Consistent**: system invariants preserved, e.g., if there were n dollars in a bank before a transfer transaction then there will be n dollars in the bank after the transfer.
- **Isolated**: Two transactions do not interfere with each other. They appear as serial executions.
- **Durable**: The commit causes a permanent change.

Recall The Synchronized Keyword

```
private double balance;

public synchronized void deposit(double amount) throws
        RemoteException {
            add amount to the balance
}

public synchronized void withdraw(double amount) throws
        RemoteException {
            subtract amount from the balance
}
```

These operations are atomic.

If one thread invokes a method it acquires a lock. Another thread will be blocked until the lock is released.

This is all that is required for many applications.

Communicating Threads

Consider a shared queue and two operations:

```
synchronized first() { removes from front }
synchronized append() { adds to rear }
```

Is this sufficient? No. If the queue is empty the client of first() will have to poll on the method. It is also potentially unfair.

Communicating Threads

Consider again the shared queue and two operations:

```
synchronized first() {
    if queue is empty call wait()
    remove from front
}
synchronized append() {
    adds to rear
    call notify()
}
```

When threads can synchronize their actions on an object by means of *wait* and *notify*, the server holds on to requests that cannot immediately be satisfied and the client waits for a reply until another client has produced whatever they need.

Note that both methods are synchronized. Only one thread at a time is allowed in.

This is general. It can get tricky fast.

Back to Transactions

- A client may require that a sequence of separate requests to a single server be atomic.
  - Free from interference from other concurrent clients.
  - Either all of the operations complete successfully or they have no effect at all in the presence of server crashes.

Assume Each Operation Is Synchronized

Client 1 Transaction T;

```
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);
```

Client 2 Transaction W;

```
total = a.getBalance();
total = total +
        b.getBalance();
total = total +
        c.getBalance();
```

Are we OK?

Assume Each Operation Is Synchronized

Client 1 Transaction T;

a.withdraw(100);

b.deposit(100);

c.withdraw(200);

b.deposit(200);

Client 2 Transaction W;

total = a.getBalance();

total = total + b.getBalance();

total = total + c.getBalance();

Inconsistent retrieval!

Assume Each Operation Is Synchronized

Client 1 Transaction T;

```
bal = b.getBalance();
b.setBalance(bal*1.1);
```

Client 2 Transaction W;

```
bal = b.getBalance();
b.setBalance(bal*1.1);
```

Are we OK?

Assume Each Operation Is Synchronized

Client 1 Transaction T;

bal = b.getBalance()

b.setBalance(bal*1.1);

Client 2 Transaction W;

bal = b.getBalance();

b.setBalance(bal*1.1);

Lost Update!

Assume Each Operation Is Synchronized

Transaction T;

a.withdraw(100);

b.deposit(100);

c.withdraw(200);

b.deposit(200);

The aim of any server that supports transactions is to maximize concurrency. So, transactions are allowed to execute concurrently if they would have the same effect as serial execution.

Each transaction is created and managed by a coordinator.

Example

Transaction T

tid = openTransaction();

    a.withdraw(tid,100);

    b.deposit(tid,100);

    c.withdraw(tid,200);

    b.deposit(tid,200);

closeTransaction(tid) or

abortTransaction(tid)

Coordinator Interface:

openTransaction() -> transID

closeTransaction(transID) ->
  commit or abort

abortTransaction(TransID)

Transaction Life Histories

Successful	Client Aborts	Server Aborts
openTransaction	openTransaction	openTransaction
operation	operation	operation
operation	operation	operation
:	:	:
operation	operation	:
closeTransaction	abortTransaction	closeTransaction returns an abort from server

Locks

- A lock is a variable associated with a data item and describes the status of that item with respect to possible operations that can be applied to that item.
- Generally, there is one lock for each item.
- Locks provide a means of synchronizing the access by concurrent transactions to the items.
- The server sets a lock, labeled with the transaction identifier, on each object just before it is accessed and removes these locks when the transaction has completed. Two types of locks are used: read locks and write locks. Two transactions may share a read lock.

This is called two phase locking.

Example: Binary Lock (1)

```
Lock_Item(x)
B:  if(Lock(x) == 0)
          Lock(x) = 1
    else {
           wait until Lock(x) == 0 and
           we are woken up.
           GOTO B
    }
```

Now, a transaction is free to use x.

Not interleaved with other code until this terminates or waits. In java, this would be a synchronized method.

Example: Binary Lock(2)

The transaction is done using x.

Unlock_Item(x)
    Lock(x) = 0
    if any transactions are waiting then
    wake up one of the waiting
    transactions.

Not interleaved with other code. If this were java, this method would be synchronized.

Locks Are Often Used To Support Concurrent Transactions

Transaction T_1	Transaction T_2
Lock_Item(x)	Lock_Item(y)
T_1 uses x	T_2 uses y
Unlock_Item(x)	Unlock_Item(y)

Think of these as remote procedure calls being executed concurrently.

In reality, the coordinator would do the locking.

If x differs from y these two transactions proceed concurrently.
If both want to use x, one waits until the other completes.

Locks May Lead to Deadlock

Four Requirements for deadlock:

(1) Resources need mutual exclusion. They are not thread safe.
(2) Resources may be reserved while a process is waiting for more.
(3) Preemption is not allowed. You can't force a process to give up a resource.
(4) Circular wait is possible. X wants what Y has and Y wants what Z has but Z wants what X has.

Solutions (short course):

Prevention (disallow one of the four)
Avoidance (study what is required by all before beginning)
Detection and recovery (reboot if nothing is getting done)

Deadlock

Transaction *T*		Transaction *U*	
Operations	Locks	Operations	Locks
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s	*a.withdraw(200);*	waits for *T*'s
	lock on *B*	•••	lock on *A*
•••		•••	
•••		•••	

Source: G. Coulouris et al., *Distributed Systems: Concepts and Design, Third Edition.*

Transactions May Be Needed on More than One Server

Begin transaction BookTrip

book a plane from Qantas

book hotel from Hilton

book rental car from Hertz

End transaction BookTrip

The Two Phase Commit Protocol is a classic solution.

Client Talks to a Coordinator

Any server

BookTrip
Coordinator

openTrans

Unique Transaction ID
TID

BookTrip Client

TID = openTransaction()

Different servers

BookPlane Participant

Recoverable objects needed
to book a plane

BookHotel Participant

Recoverable objects needed
to book a hotel.

BookRentalCar Participant

Recoverable objects needed
to rent a car.

Client Uses Services

Different servers

Any server

BookTrip
Coordinator

BookPlane Participant

Recoverable objects needed to book a plane

BookHotel Participant

Recoverable objects needed to book a hotel.

Call + TID

BookRentalCar Participant

BookTrip Client

Recoverable objects needed to rent a car.

plane.bookFlight(111,"Seat32A",TID)

Participants Talk to Coordinator

Different servers

The participant only calls join if it has not already done so.

BookPlane Participant

Recoverable objects needed to book a plane

BookHotel Participant

Recoverable objects needed to book a hotel.

BookRentalCar Participant

BookTrip Coordinator

join(TID,ref to participant)

BookTrip Client

The participant knows where the coordinator is because that information can be included in the TID (eg. an IP address.)
The coordinator now has a pointer to the participant.

Suppose All Goes Well (1)

BookTrip Coordinator

<u>Different servers</u>

BookPlane Participant

Recoverable objects needed to book a plane

BookHotel Participant

Recoverable objects needed to book a hotel.

BookRentalCar Participant

Recoverable objects needed to rent a car.

BookTrip Client

OK returned

OK returned

OK returned

Suppose All Goes Well (2)

Different servers

BookTrip Coordinator

Coordinator begins 2PC and this results in a GLOBAL COMMIT sent to each participant.

BookPlane Participant

Recoverable objects needed to book a plane

BookHotel Participant

Recoverable objects needed to book a hotel.

BookRentalCar Participant

Recoverable objects needed to rent a car.

BookTrip Client

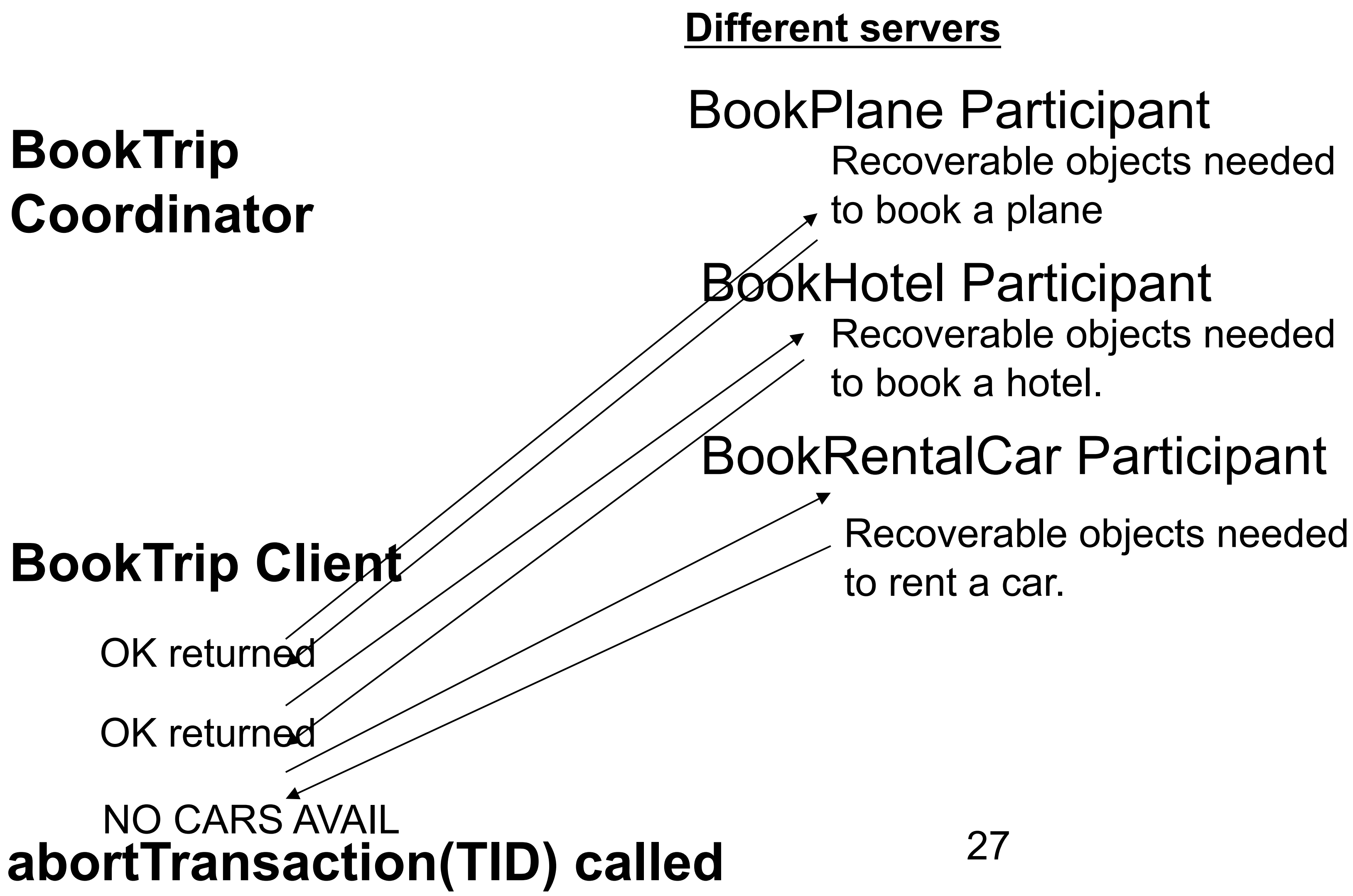
OK returned

OK returned

OK returned

CloseTransaction(TID) Called

This Time No Cars Available (1)



This Time No Cars Available (2)

Different servers

BookPlane Participant

Recoverable objects needed to book a plane

BookHotel Participant

Recoverable objects needed to book a hotel.

BookRentalCar Participant

Recoverable objects needed to rent a car.

BookTrip Coordinator

Coordinator sends a GLOBAL_ABORT to all particpants

BookTrip Client

OK returned

OK returned

NO CARS AVAIL

abortTransaction(TID) called

This Time No Cars Available (3)

Different servers

BookTrip Coordinator

abortTransaction

Each participant
Gets a GLOBAL_ABORT

BookPlane Participant
ROLLBACK CHANGES

BookHotel Participant
ROLLBACK CHANGES

BookRentalCar Participant
ROLLBACK CHANGES

BookTrip Client

OK returned

OK returned

NO CARS AVAIL

abortTransaction(TID)

BookPlane Server Crashes After Returning 'OK' (1)

BookTrip Coordinator

BookTrip Client

Different servers

BookPlane Participant

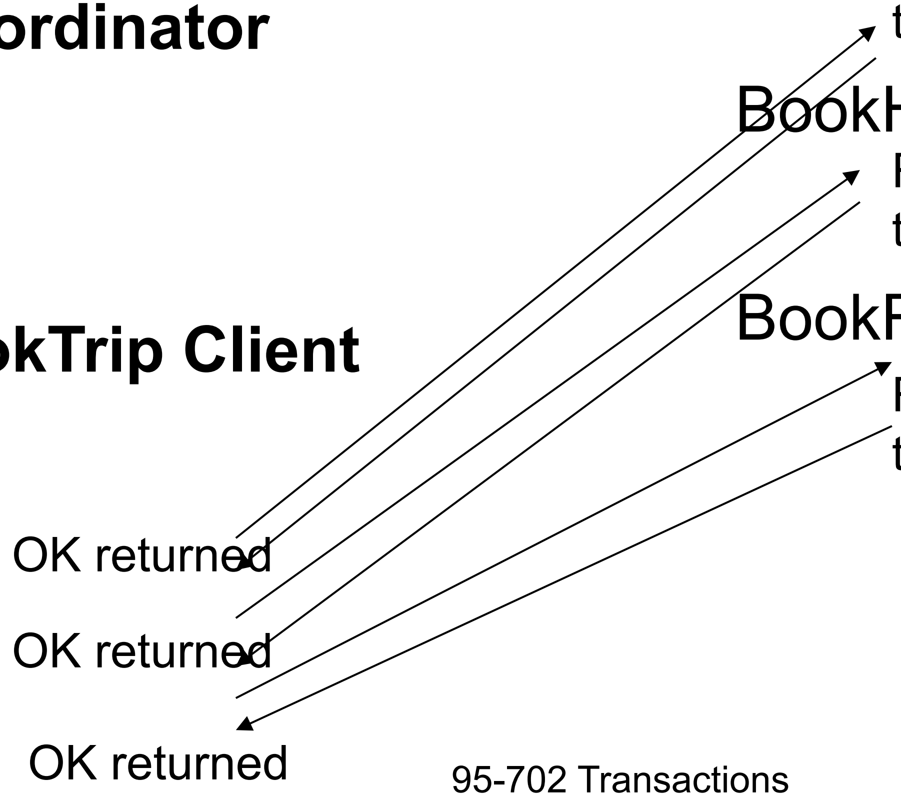
Recoverable objects needed to book a plane

BookHotel Participant

Recoverable objects needed to book a hotel.

BookRentalCar Participant

Recoverable objects needed to rent a car.



BookPlane Server Crashes After Returning 'OK' (2)

BookTrip Coordinator

Coordinator excutes 2PC:
Ask everyone to vote.
No news from the BookPlane Participant so multicast a GLOBAL ABORT

BookTrip Client

OK returned

OK returned

OK returned

CloseTransaction(TID) Called

Different servers

~~BookPlane Participant~~

~~Recoverable objects needed to book a plane~~

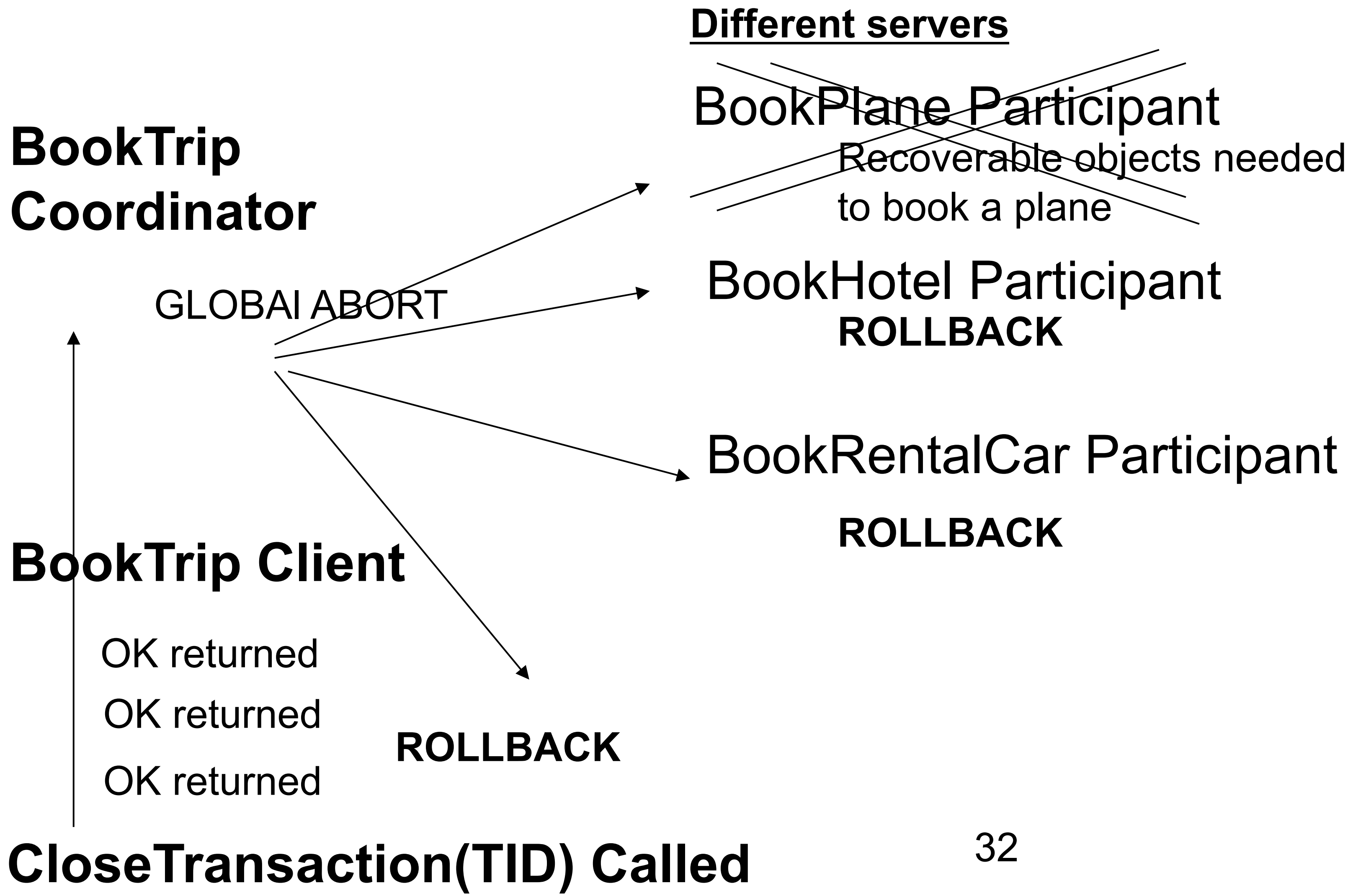
BookHotel Participant

Recoverable objects needed to book a hotel.

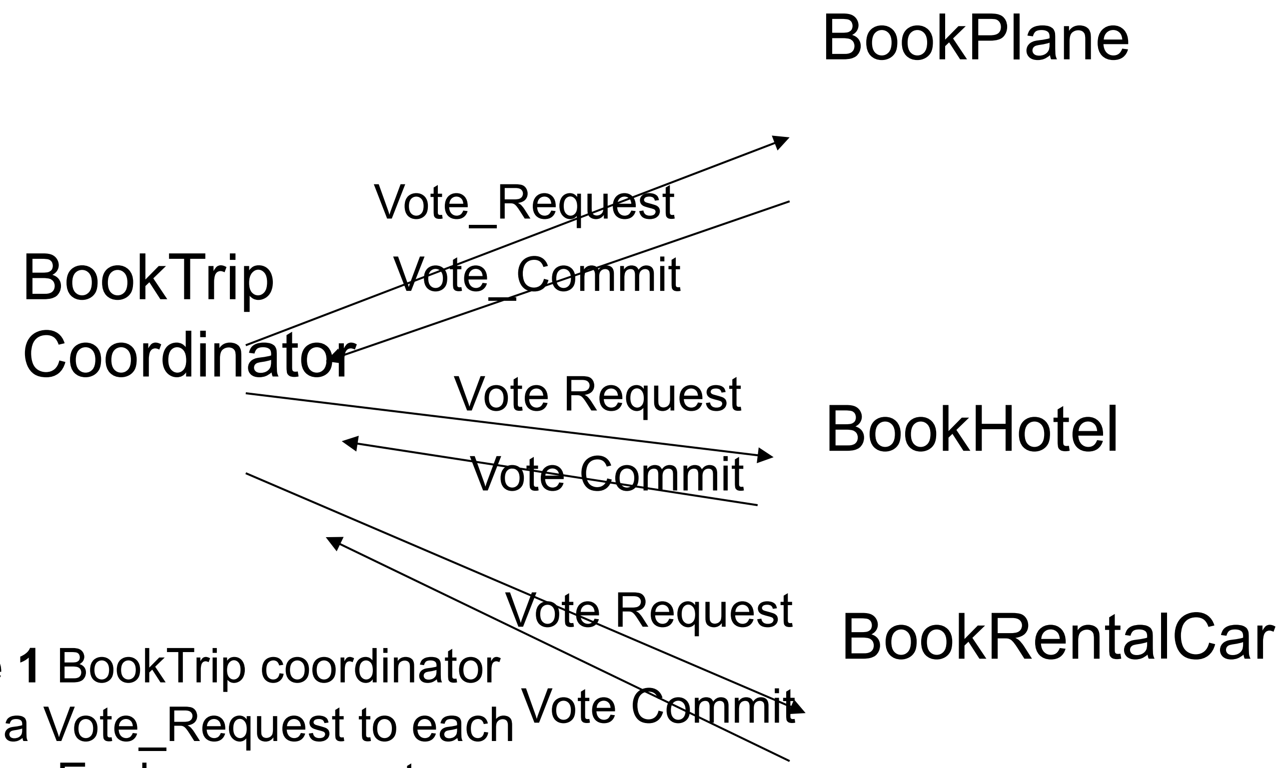
BookRentalCar Participant

Recoverable objects needed to rent a car.

BookPlane Server Crashes after returning 'OK' (3)

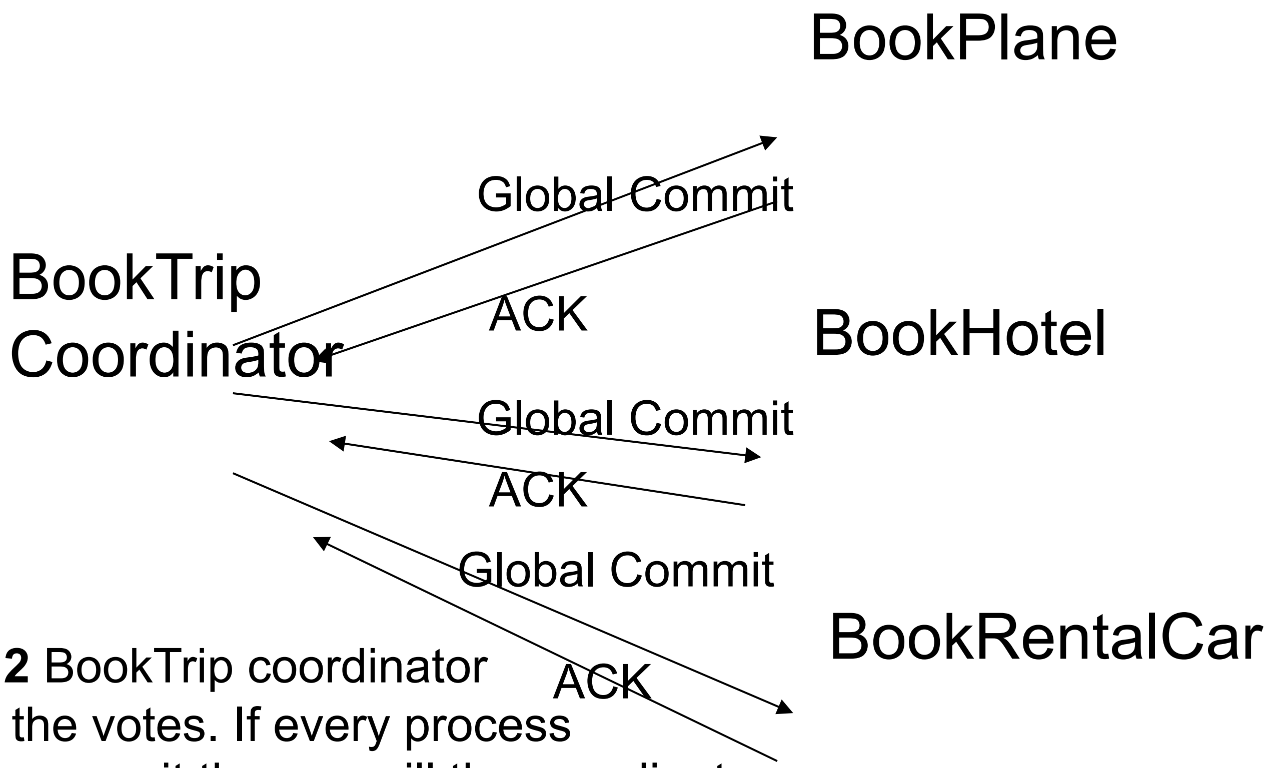


Two-Phase Commit Protocol



Phase 1 BookTrip coordinator sends a Vote_Request to each process. Each process returns a Vote_Commit or Vote_Abort.

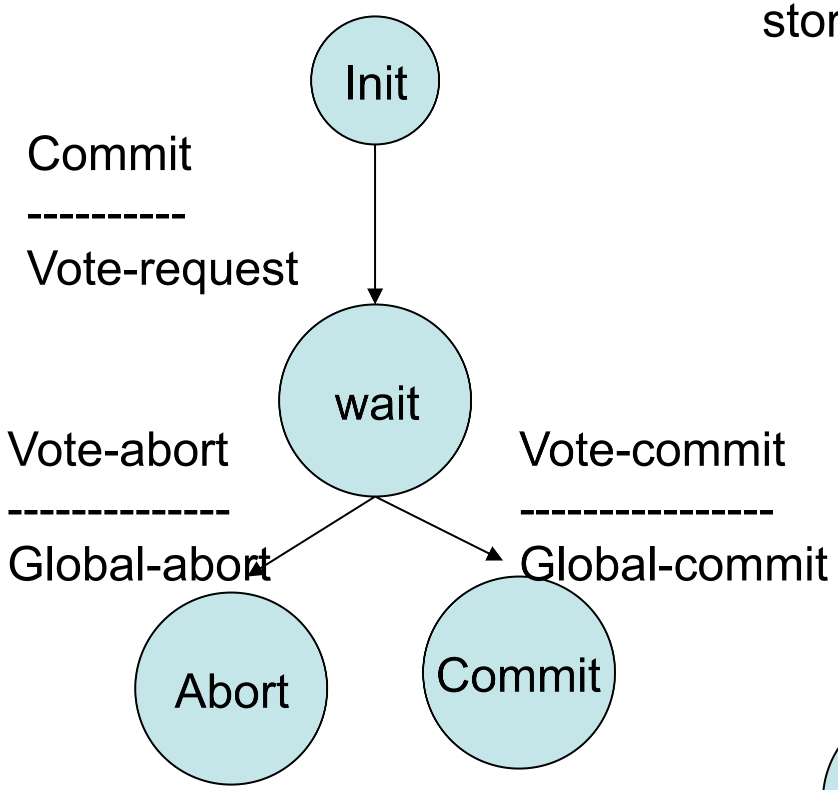
Two-Phase Commit Protocol



Phase 2 BookTrip coordinator checks the votes. If every process votes to commit then so will the coordinator.
In that case, it will send a Global_Commit to each process.
If any process votes to abort the coordinator sends a GLOBAL_ABORT.
Each process waits for a Global_Commit message before committing its part of the transaction.

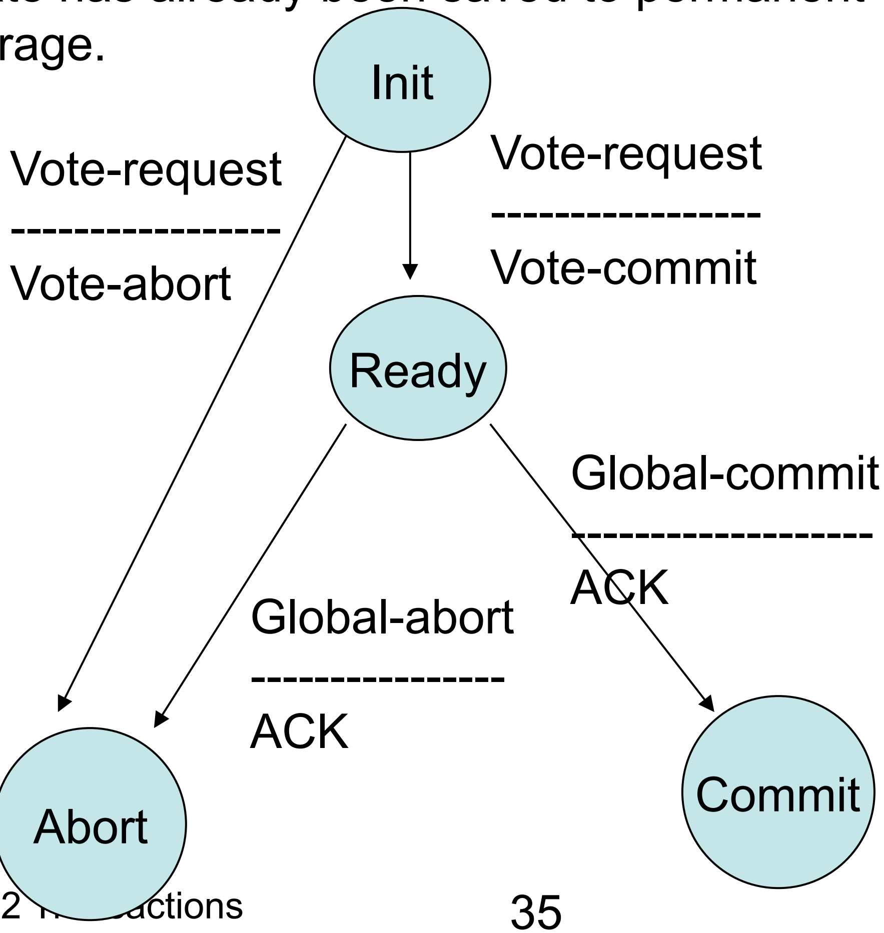
2PC Finite State Machine from Tanenbaum

BookTrip Coordinator



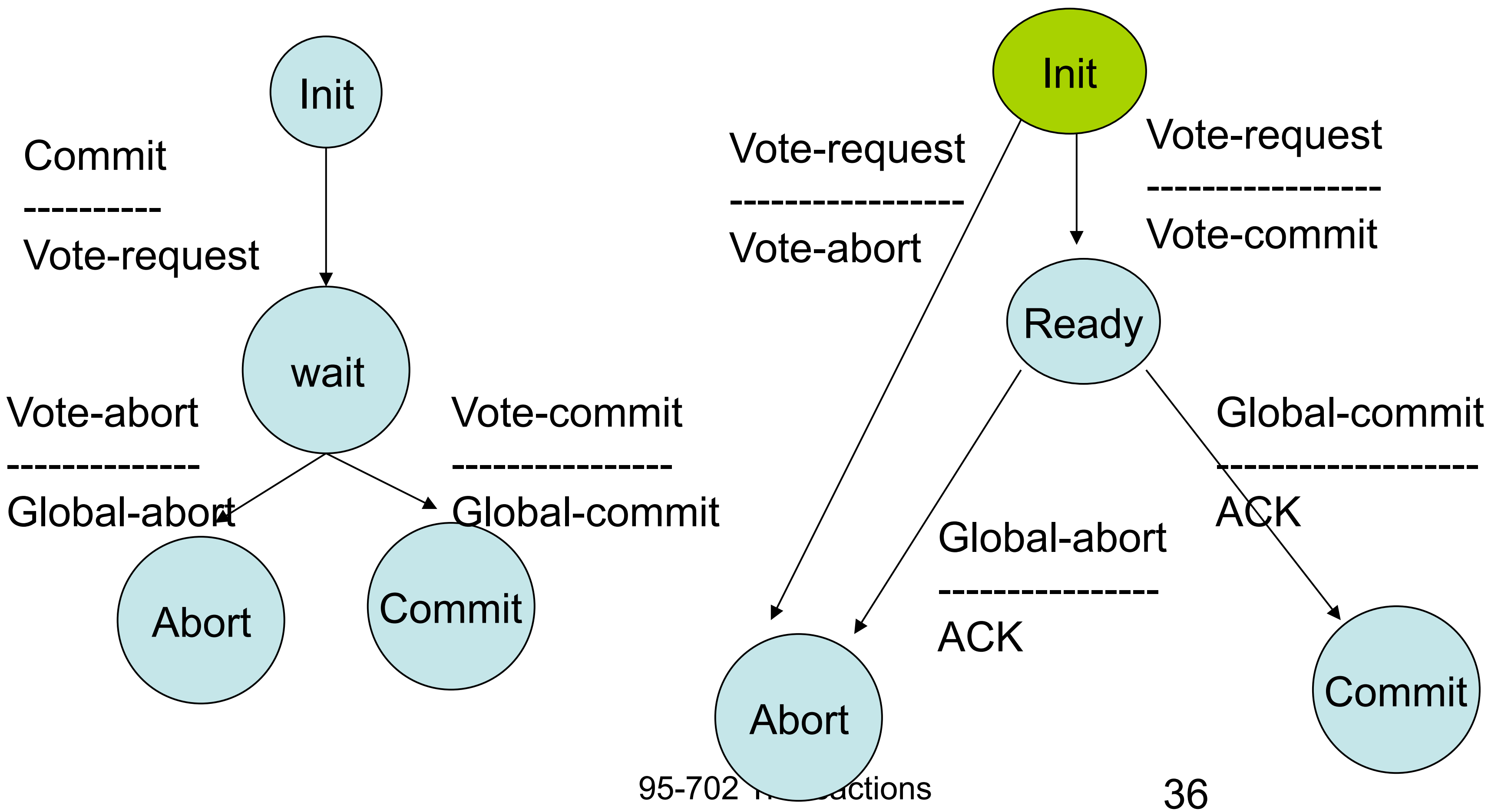
Participant

State has already been saved to permanent storage.

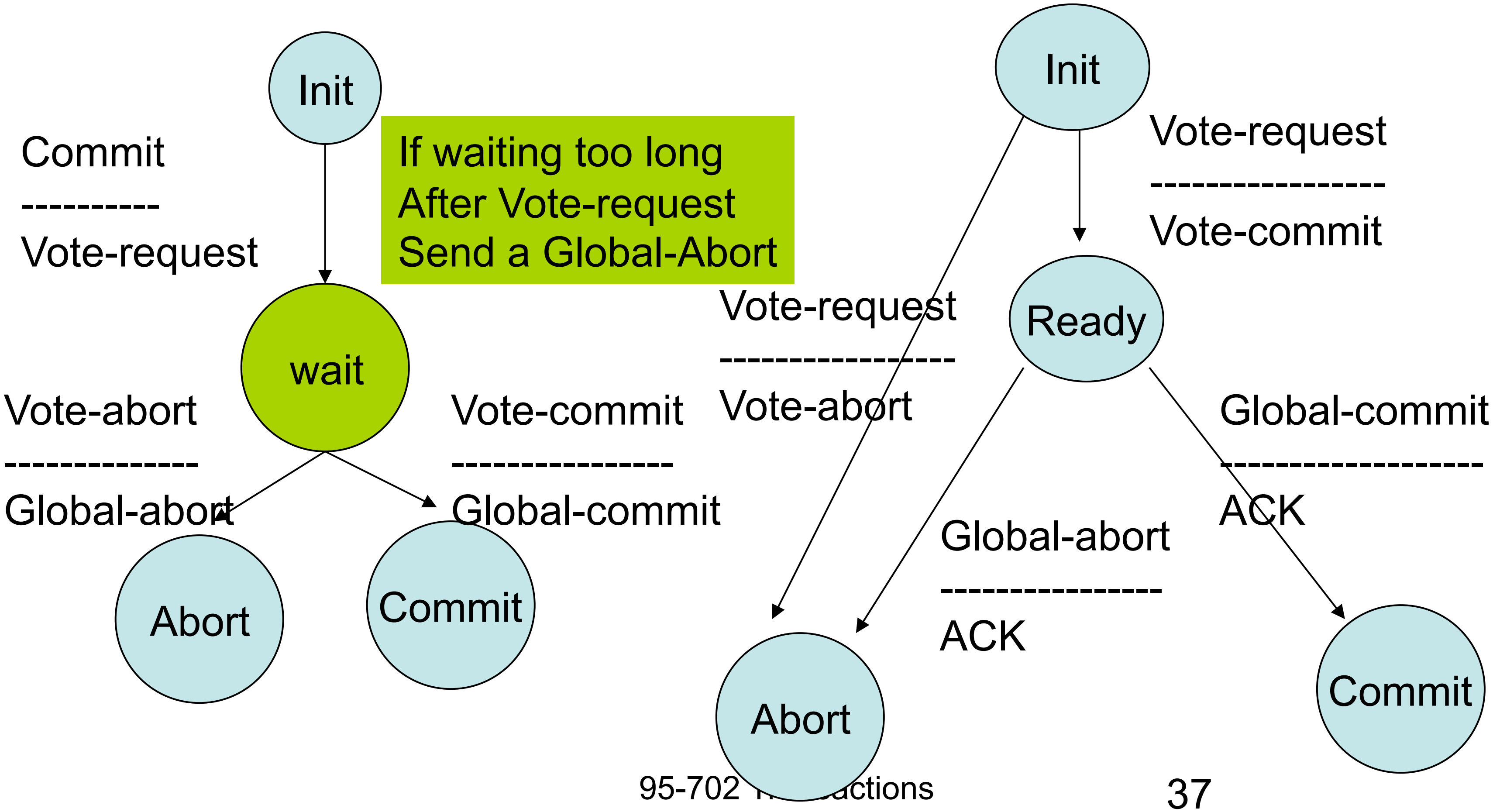


2PC Blocks in Three Places

If waiting too long for a Vote-Request send a Vote-Abort

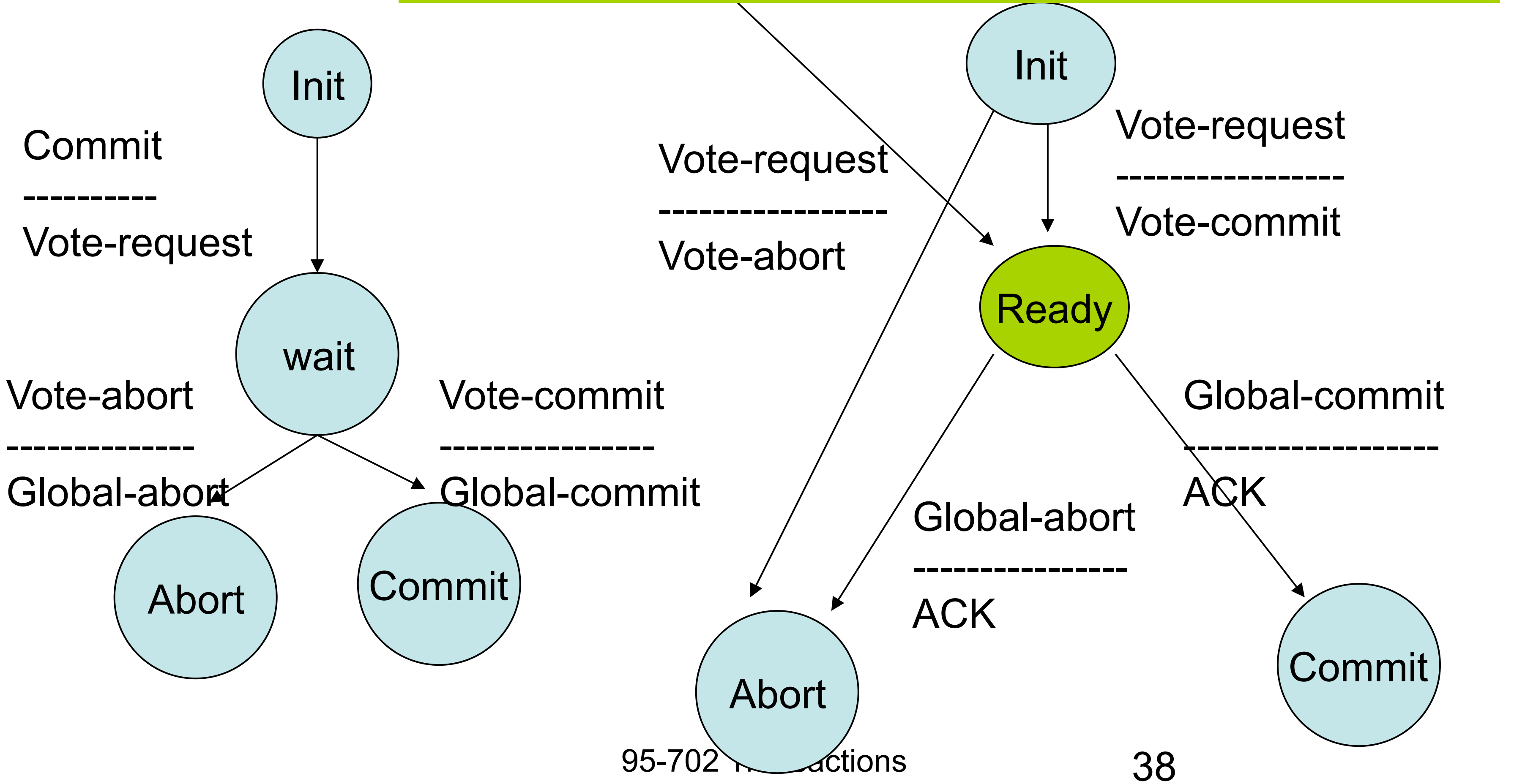


2PC Blocks in Three Places



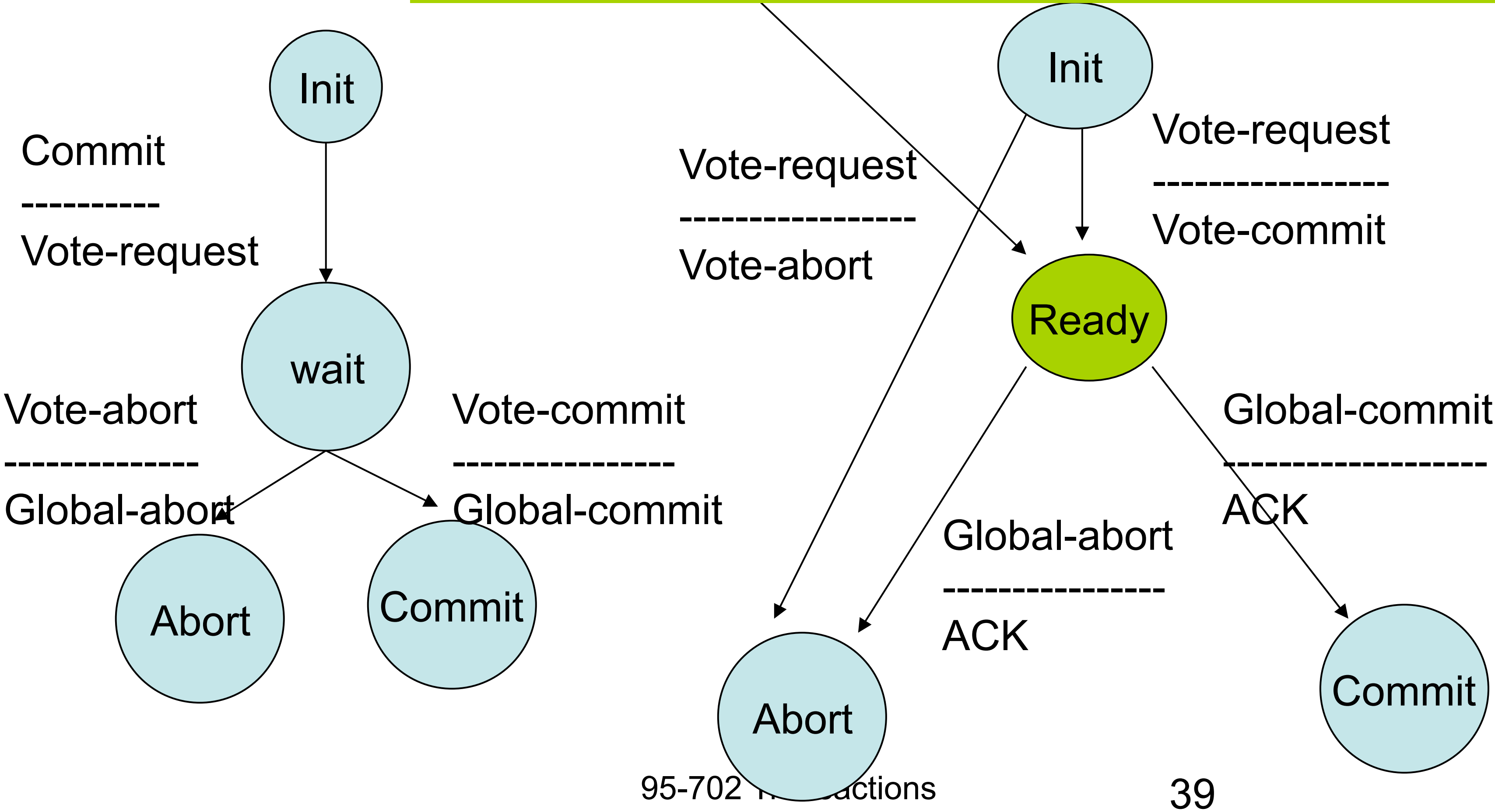
2PC Blocks in Three Places

If waiting too long we can't simply abort! We must wait until the coordinator recovers. We might also make queries on other participants.



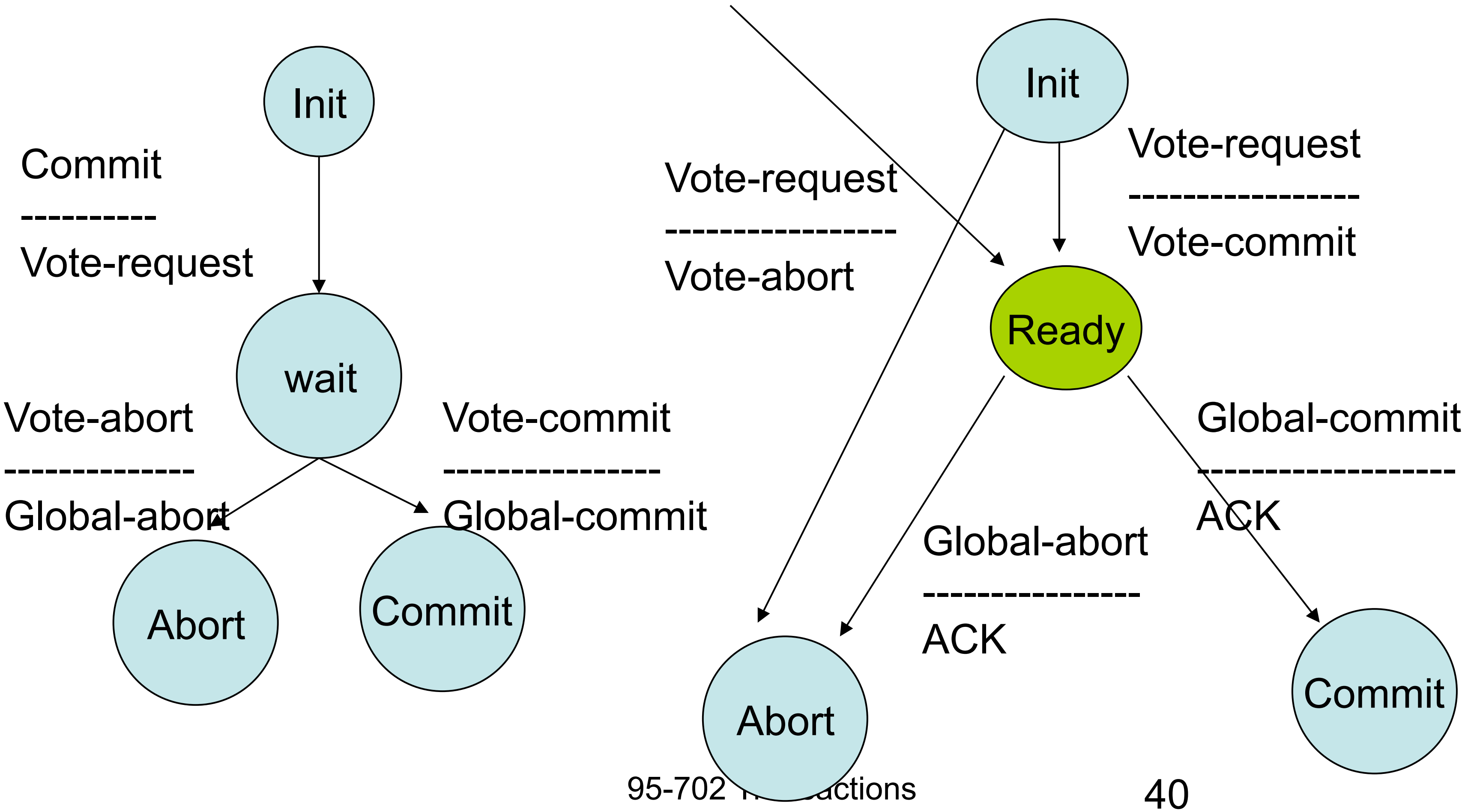
2PC Blocks in Three Places

If this process learns that another has committed then this process is free to commit. The coordinator must have sent out a Global-commit that did not get to this process.



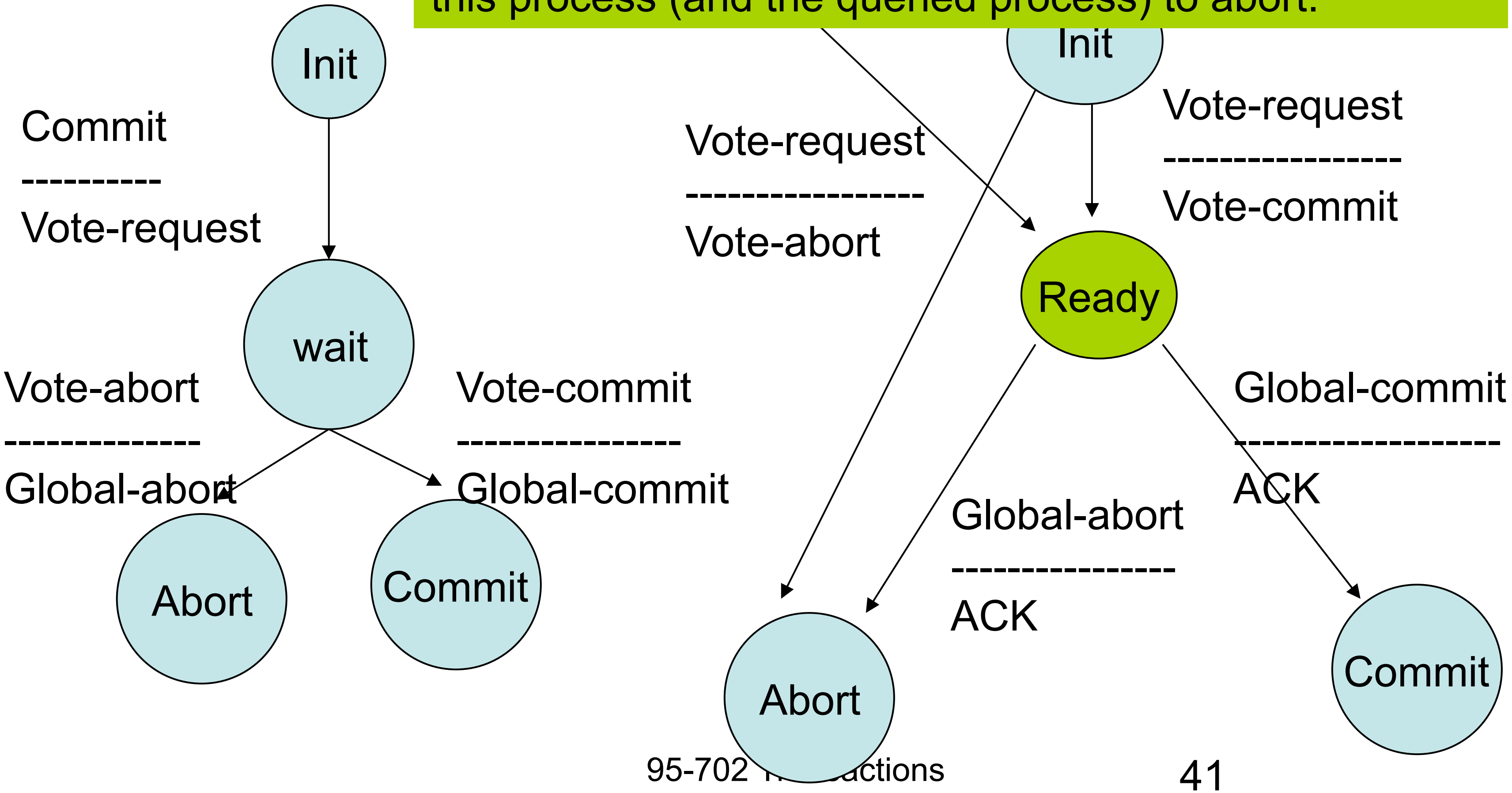
2PC Blocks in Three Places

If this process learns that another has aborted then it too is free to abort.



2PC Blocks in Three Places

Suppose this process learns that another process is still in its init state. The coordinator must have crashed while multicasting the Vote-request. It's safe for this process (and the queried process) to abort.



2PC Blocks in Three Places

Tricky case: If the queried processes are all still in their ready state what do we know? We have to block and wait until the Coordinator recovers.