Sistemas Distribuídos

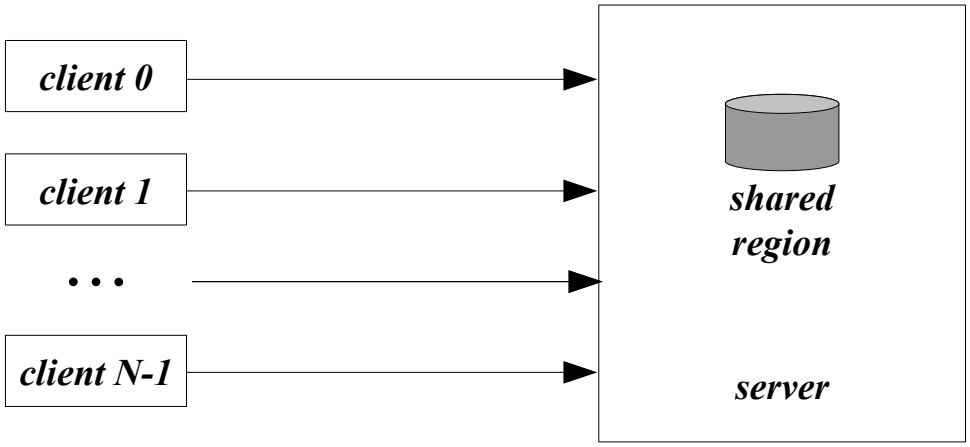
Distributed Transactions

António Rui Borges

Summary

- *What is a transaction?*
  - *Characterization*
- *Organization of operations*
  - *Concurrency issues*
- *Distributed transactions*
  - *Two stage commitment*
  - *Three stage commitment*
- *Suggested reading*

What is a transaction? - 1



A *transaction* can be thought of a set of *read* like and *write* like operations requested by a client to be carried out in registers of a shared region that the server, which manages it, must perform as an indivisible unit.

Thus, the server has to ensure that the results of the set of operations are saved as a whole in permanent storage, or dismissed as a whole.

What is a transaction? - 2

The properties of transactions may be described by the acronym ACID (Härder and Reuter)

- *atomicity* – a transaction must be all or nothing
- *consistency* – a transaction must take the shared region from one consistent state to another
- *isolation* – a transaction must be performed without any interference from another, that is, the intermediate effects of a transaction can not be seen by other transactions that are taking place at the same time
- *durability* – once having been carried out successfully, the effects generated by a transaction must last forever.

Organization of operations - 1

A transaction is created and managed by a *coordinator* process on the server side. Upon client request, a transaction is opened by the *coordinator* process and a *transaction id* is assigned to it.

All the following *read* like and *write* like operations on specific registers issued by the client must have the *transaction id* attached to it and return the operation *status*: either the operation succeeded and the client may go on with further operations, or the whole transaction is aborted. The client itself may at any time abort the transaction it is being run on its behalf.

To complete the transaction, the client issues an end of transaction command. If the status is successful, the whole set of *read* like and *write* like operations are *commited* and the changes produced on the addressed registers of the shared region become permanent; otherwise, an abort status is returned.

When a transaction is aborted, the client may try it again later on.

Organization of operations - 2

Successful transaction	**Aborted transaction (by the client)**	**Aborted transaction (by the server)**	**Aborted transaction (by the server)**
openTransaction	openTransaction	openTransaction	openTransaction
operation 1	operation 1	operation 1	operation 1
operation 2			operation 2
…	…	…	…
…	…	operation K (abort)	…
…	abortTransaction		…
operation N			operation N
closeTransaction (success)			closeTransaction (abort)

Concurrency issues - 1

Having multiple transactions performing operations on the same registers of the shared region potentially at the same time, racing conditions become an issue and have to be dealt with in a controlled fashion so that inconsistencies on the data are not generated.

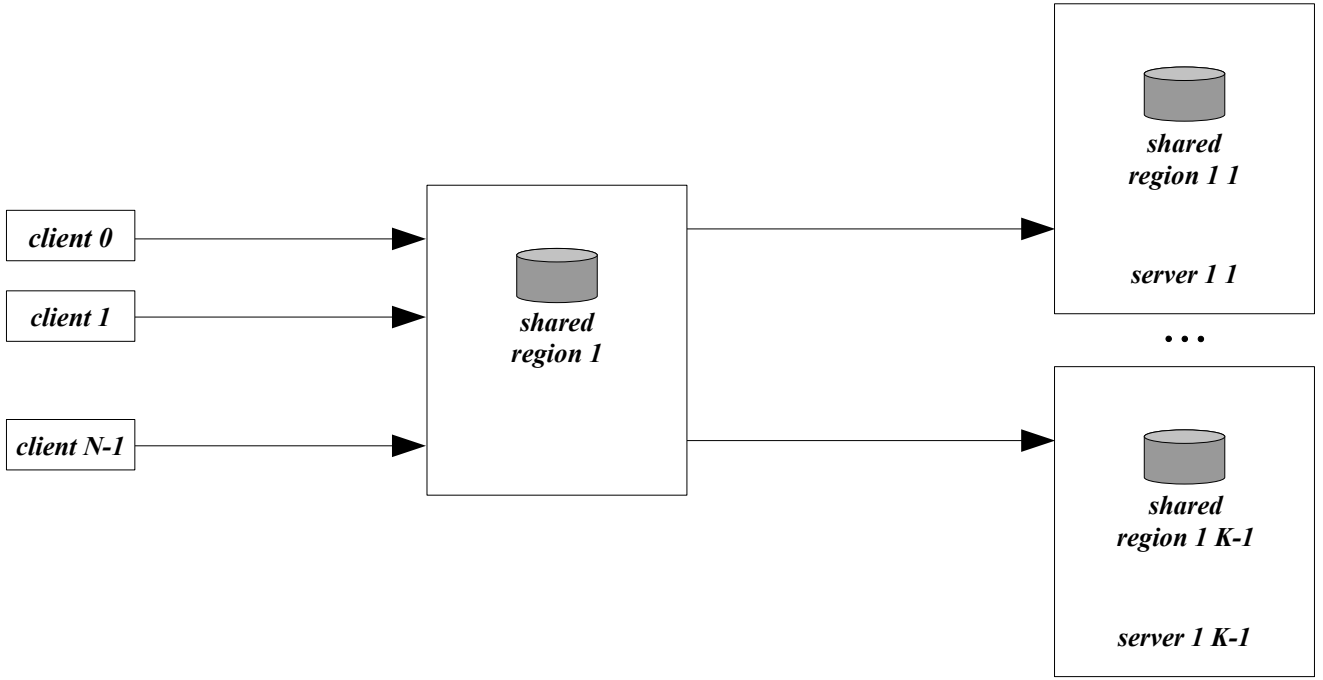
To begin with, one should consider the impact of pairs of *read* like and *write* like operations performed by two different transactions on the same register.

Transaction T1	Transaction T2	Impact
read	read	none by itself
read	write	dependent on the order of execution
write	write	dependent on the order of execution

Concurrency issues - 2

So if two transactions attempt to modify a given register, the register must be locked first by one of the transactions and then the modification is carried out in a copy. Only when the transaction completes, the register contents is updated and the lock released. Then, the second transaction may proceed.

Distributed transactions



In a *distributed transaction*, the shared region of interest is divided in multiple parts, each being run by a different server.

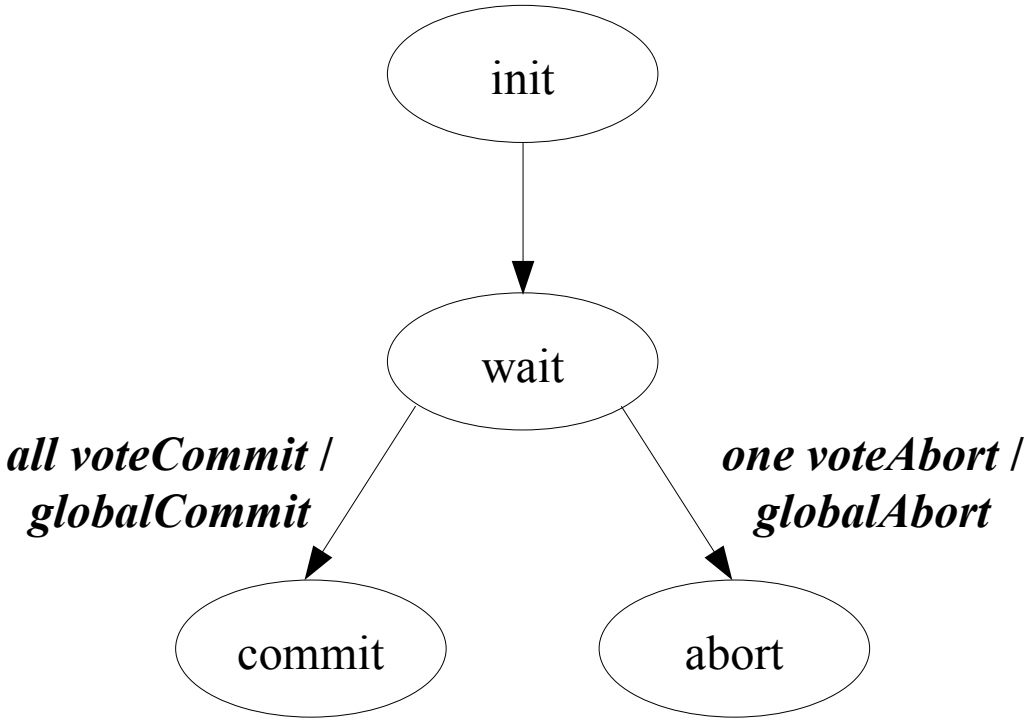
The key question that comes now is how to ensure that the results of the set of operations carried out by the different servers are saved as a whole in permanent storage, or dismissed as a whole.

Two stage commitment - 1

- the process *coordinator* process sends the message *voteRequest* to all the *participant* processes in the transaction
- when a *participant* process receives the message, it answers by sending the message
  - *voteCommit* to the *coordinator* process, if it is ready to commit its part of the transaction
  - *voteAbort* to the *coordinator* process, otherwise
- the process *coordinator* collects the *voting* messages and sends back the message
  - *globalCommit* to all the *participant* processes, if all of them have voted *voteCommit* for their part of the transaction
  - *globalAbort* to all the *participant* processes, if at least one of them has stated it can not commit its part of the transaction
- the *participant* processes wait for the confirmation of the *coordinator* process, if the received message is
  - *globalCommit*, they commit their part of the transaction
  - *globalAbort*, they discard their part of the transaction.

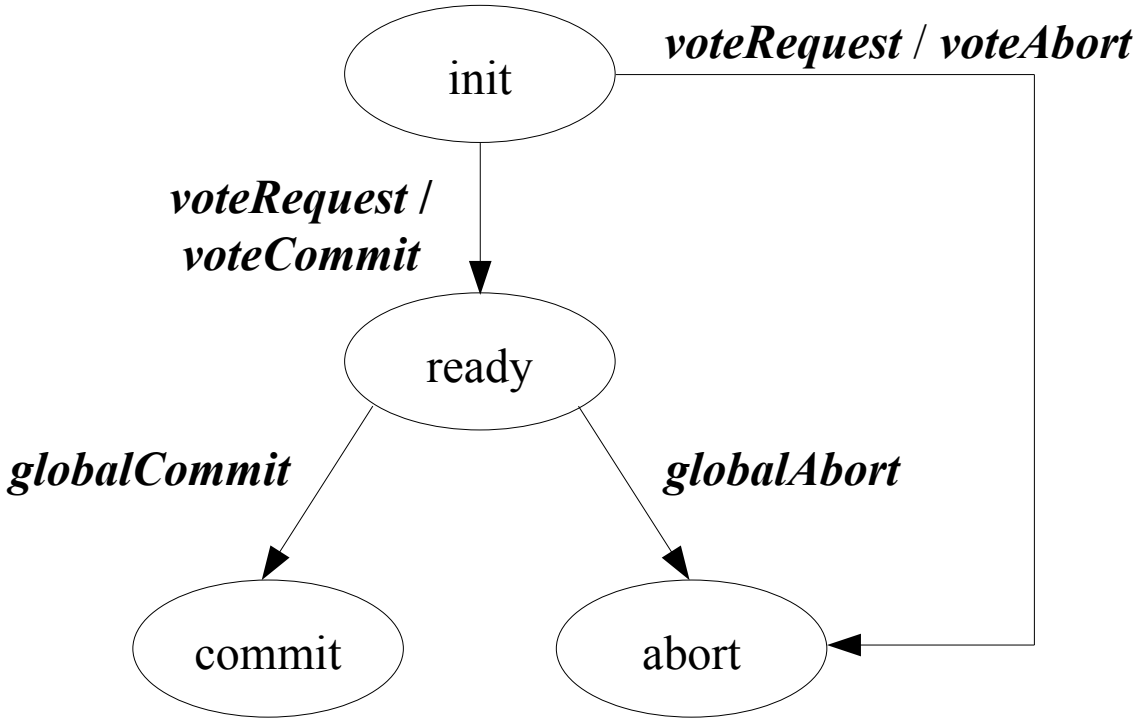
Two stage commitment - 2

coordinator process (no fails have occurred)



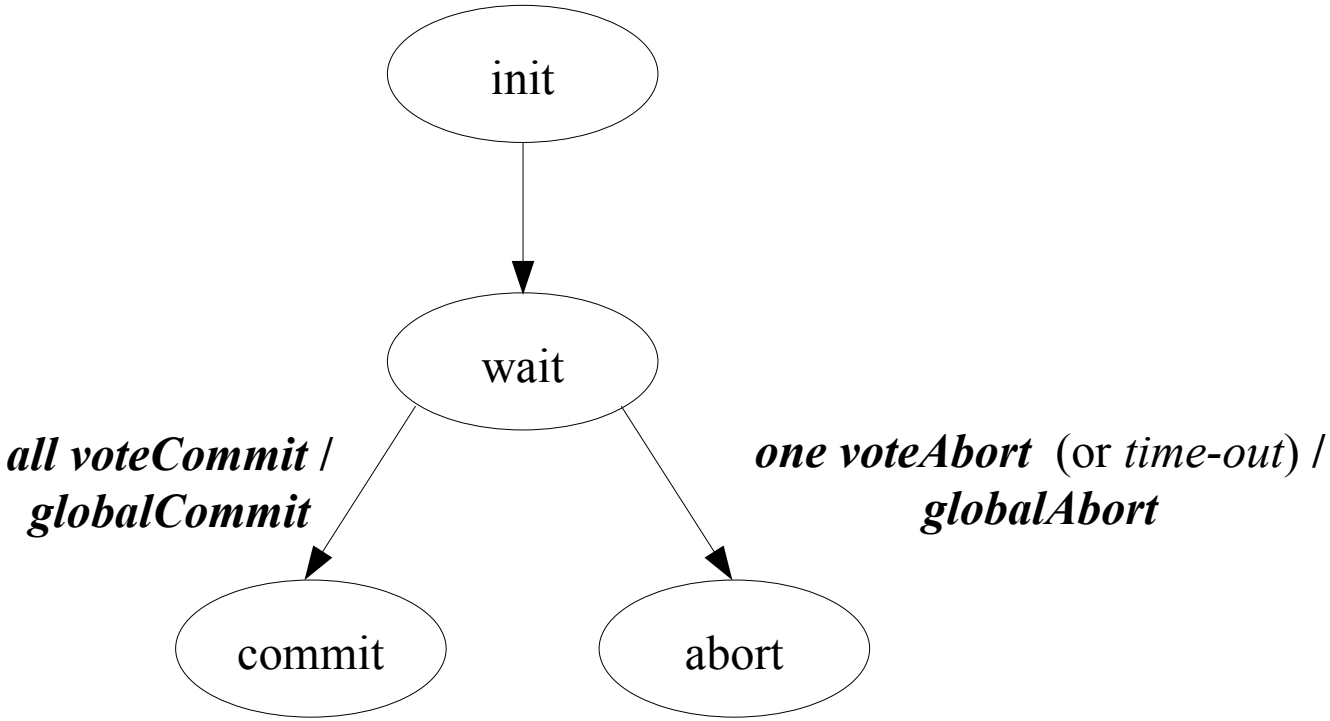
Two stage commitment - 3

participant processes (no fails have occurred)



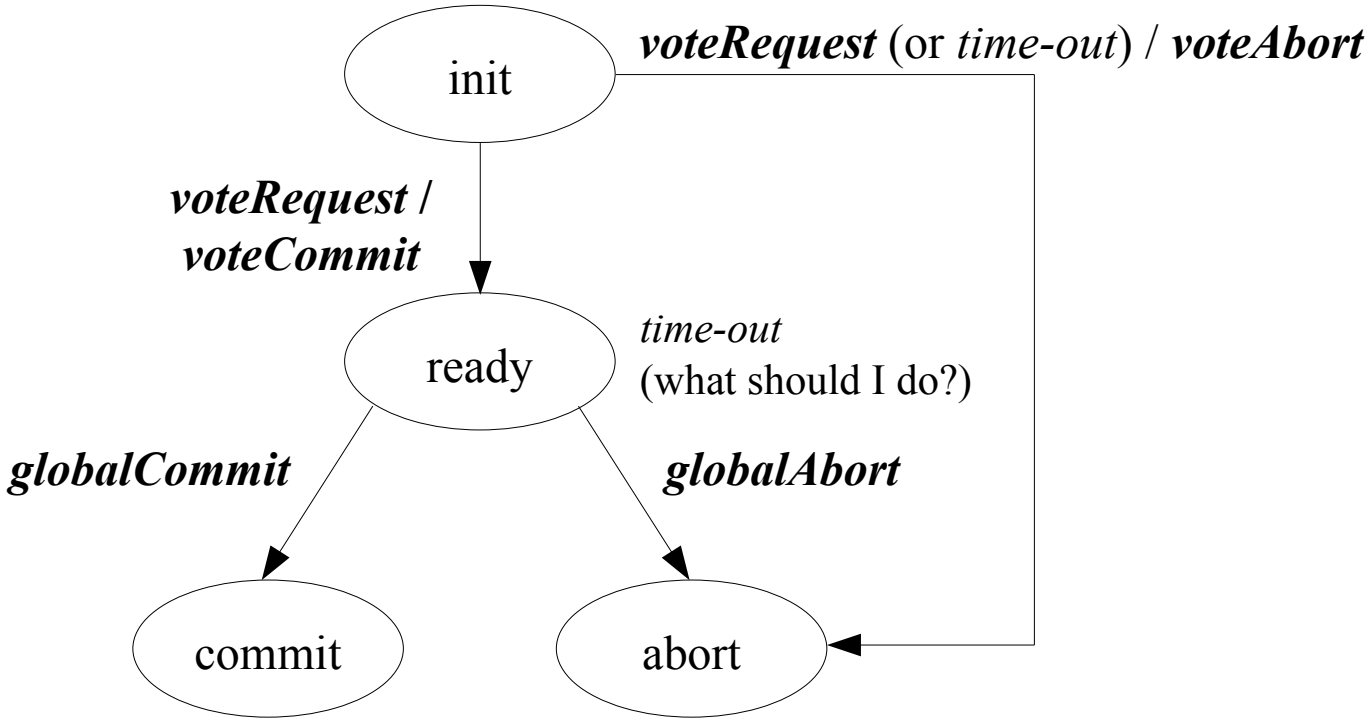
Two stage commitment - 4

coordinator process (fails have occurred)



Two stage commitment - 5

participant processes (fails have occurred)



Two stage commitment - 6

what p_i should do

state of p ($j \neq i$)	**action to be taken by p_i**
init	transit to abort
ready	contact other participant
commit	transit to commit
abort	transit to abort

Two stage commitment - 7

coordinator process

```
state = init;
writeLocalLog (state);
numberOfVotes = 0;
multicast (voteRequest) to all participant processes;
state = wait;
writeLocalLog (state);
while (numberOfVotes < N)
{ wait for (vote);
  if (timeout)
     { state = abort;
       writeLocalLog (state);
       multicast (globalAbort) to all participant processes;
       return;
     }
  writeLocalLog (vote);
}
```

Two stage commitment - 8

coordinator process (continuation)

```
if (all participants voted voteCommit)
   { state = commit;
     writeLocalLog (state);
     multicast (globalCommit) to all participant processes;
   }
   else { state = abort;
          writeLocalLog (state);
          multicast (globalAbort)to all participant processes;
        }
```

Two stage commitment - 9

participant processes

```
state = init;
writeLocalLog (state);
wait for (voteReq);
if (timeout)
    { state = abort;
      writeLocalLog (state);
      return;
    }
if (participant votes voteAbort)
    { state = abort;
      writeLocalLog (state);
      unicast (coord, voteAbort);
      return;
    }
```

Two stage commitment - 10

participant processes (continuation)

```
if (participant votes voteCommit)
   { state = ready;
     writeLocalLog (state);
     unicast (coord, voteCommit);
     wait for (decision);
     if (timeout)
        multicast (decisionRequest) to all other participant processes;
        else { writeLocalLog (decison);
               if (decision == globalCommit)
                  { state = commit;
                    writeLocalLog (state);
                    return;
                  }
                  else { state = abort;
                         writeLocalLog (state);
                         return;
                       }
             }
   }
```

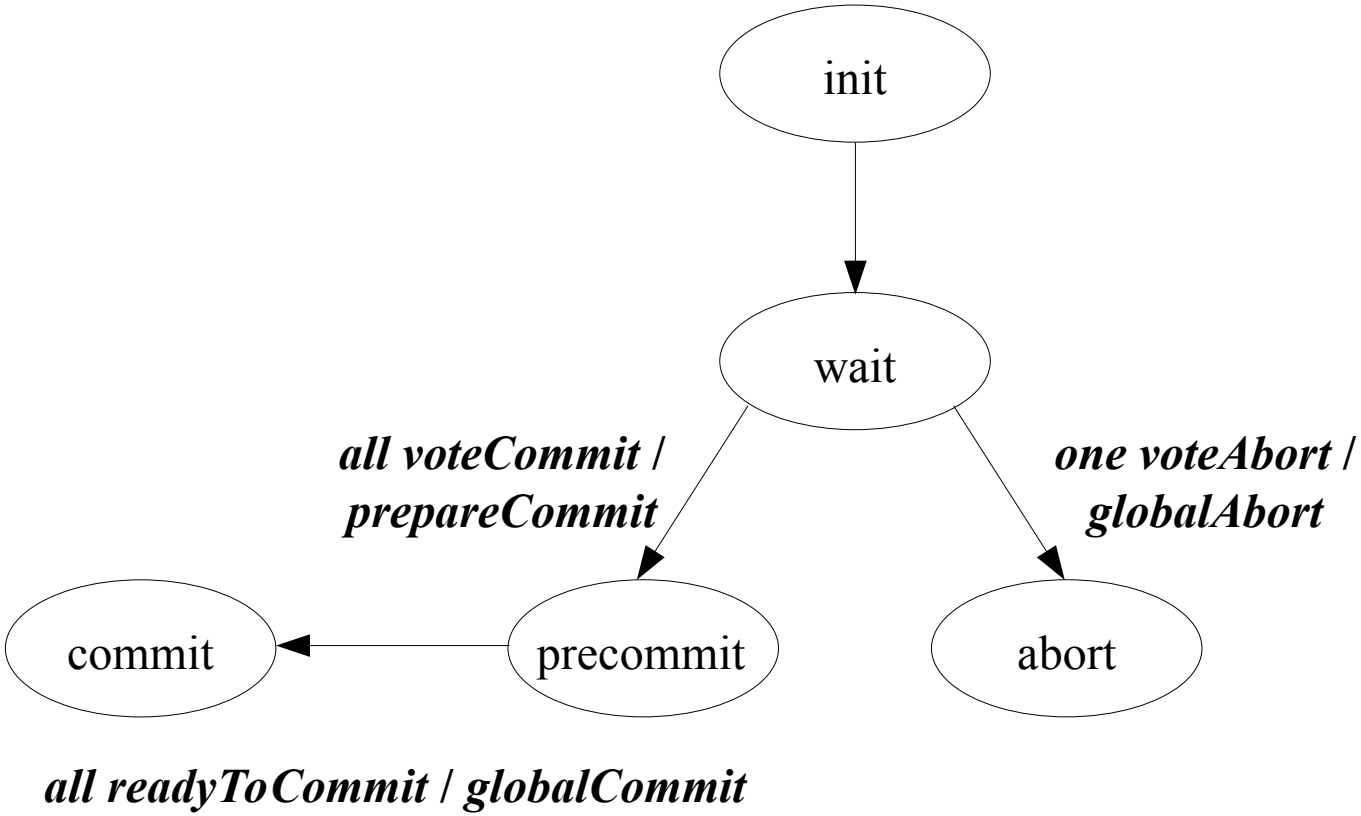
Two stage commitment - 11

participant processes (continuation)

```
while (true)
{ wait for (decisionRequest);
  if (decisionRequest == commit)
     { state = commit;
       writeLocalLog (state);
       return;
     }
     else if ((decisionRequest == init) || (decisionRequest == abort))
             { state = abort;
               writeLocalLog (state);
               return;
             }
             else block again;
}
```

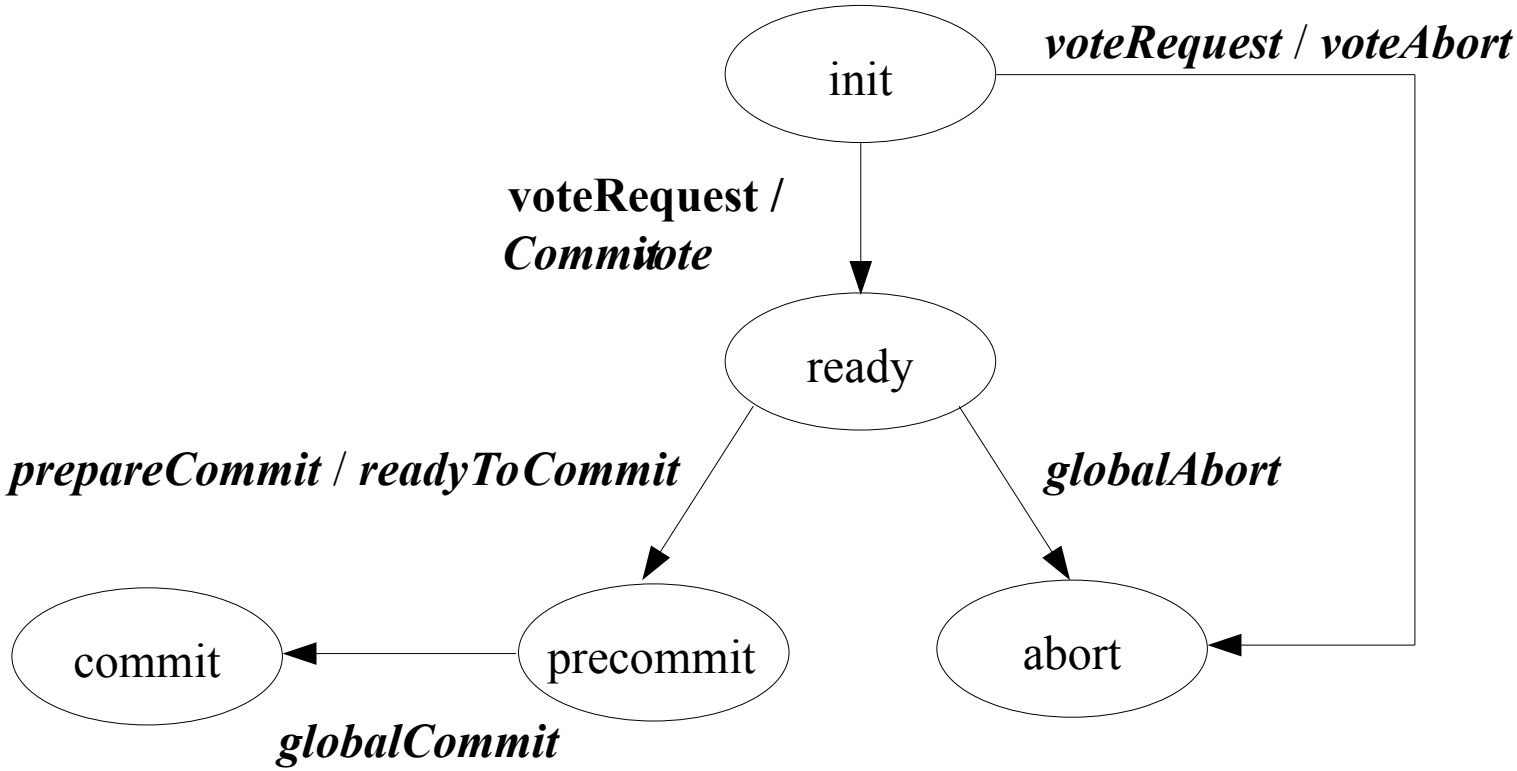
Three stage commitment - 1

coordinator process (no fails have occurred)



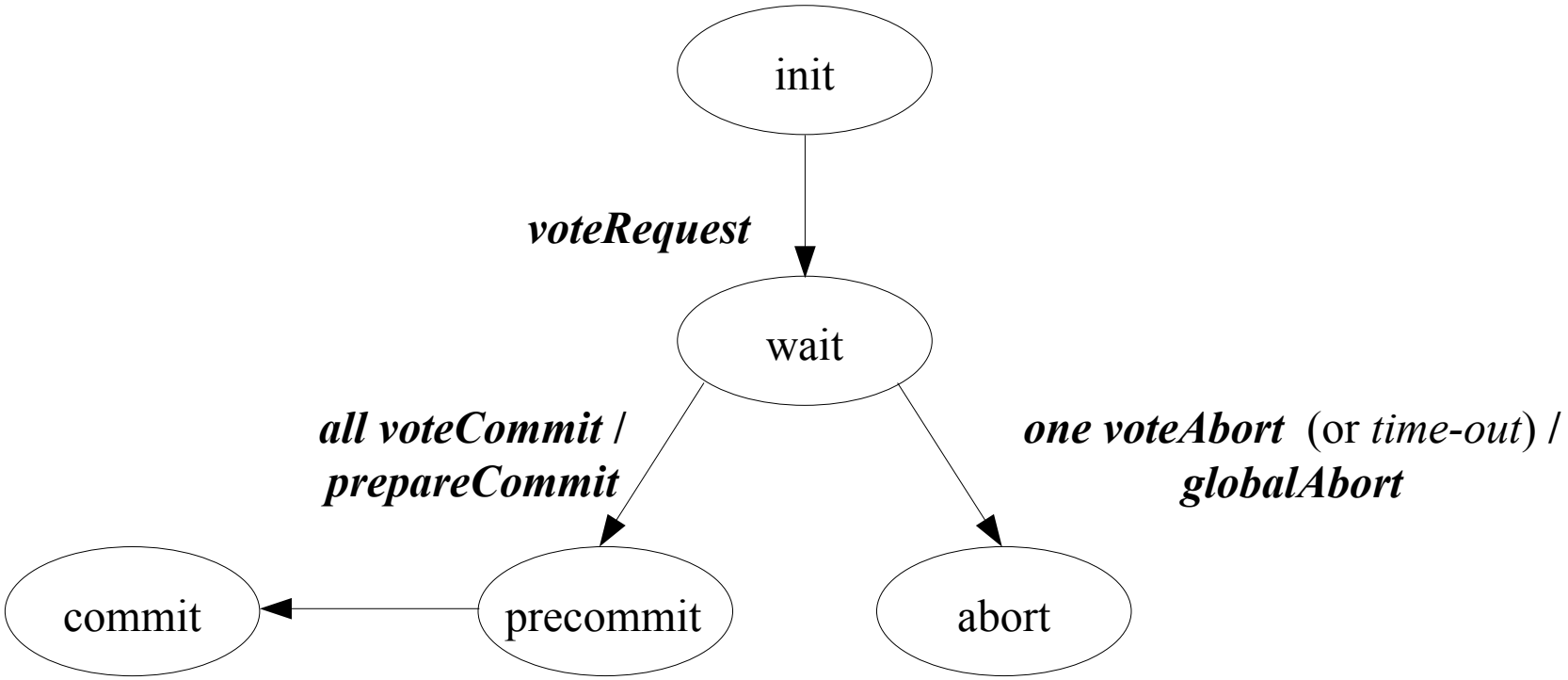
Three stage commitment - 2

participant processes (no fails have occurred)



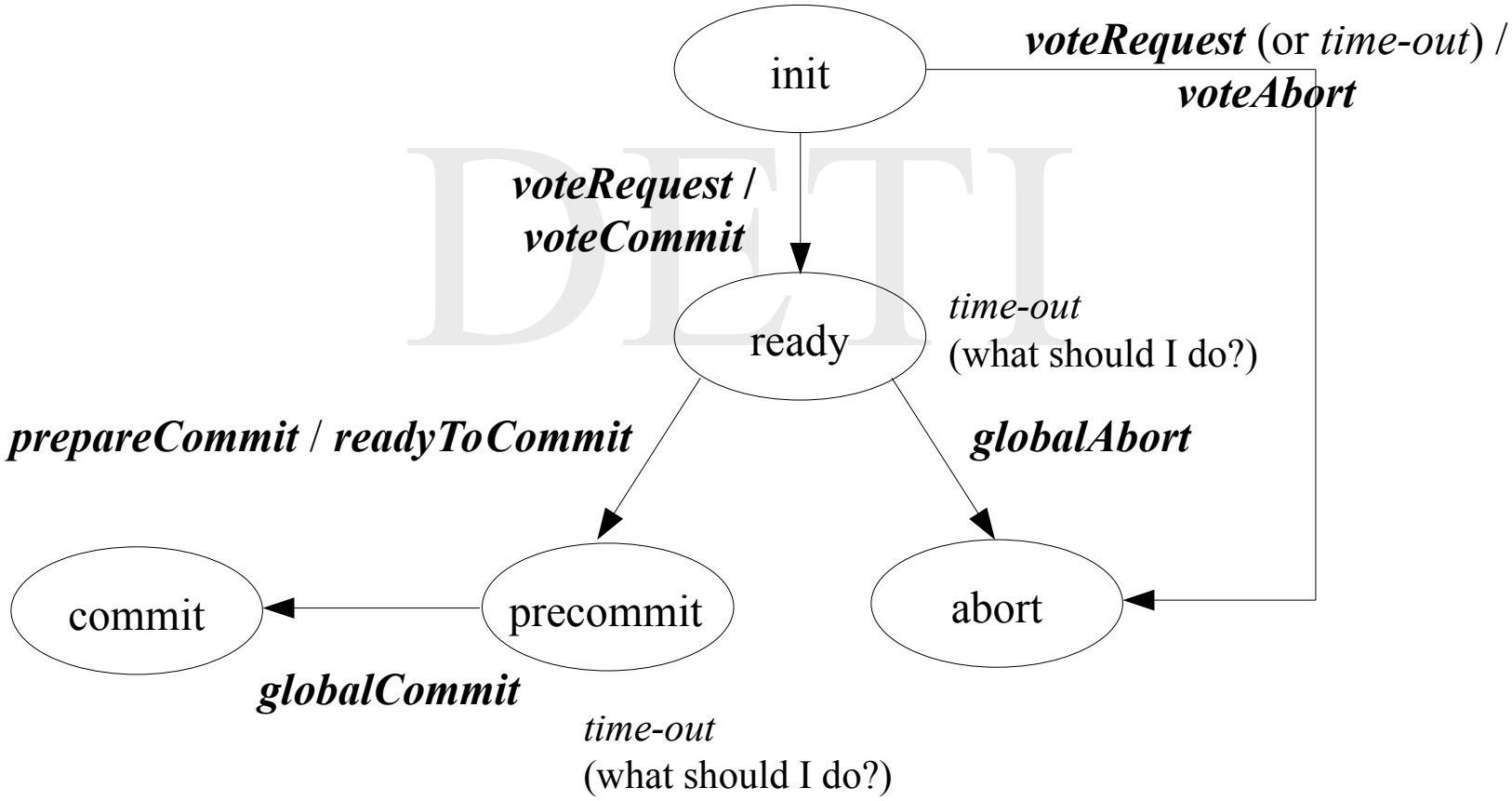
Three stage commitment - 3

coordinator process (fails have occurred)



Three stage commitment - 4

participant processes (no fails have occurred)



Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 13: *Transactions and concurrency control*
    - Sections 13.1 to 13.4
  - Chapter 14: *Distributed transactions*
    - Sections 14.1 to 14..3
- *Distributed Systems: Principles and Paradigms, 2 Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 8: *Fault tolerance*
    - Sections 8.1 and 8.5