The UNIVERSITY *of* EDINBURGH

SCHOOL *of* INFORMATICS

CS4/MSc

Distributed Systems

Björn Franke

bfranke@inf.ed.ac.uk

Room 2414

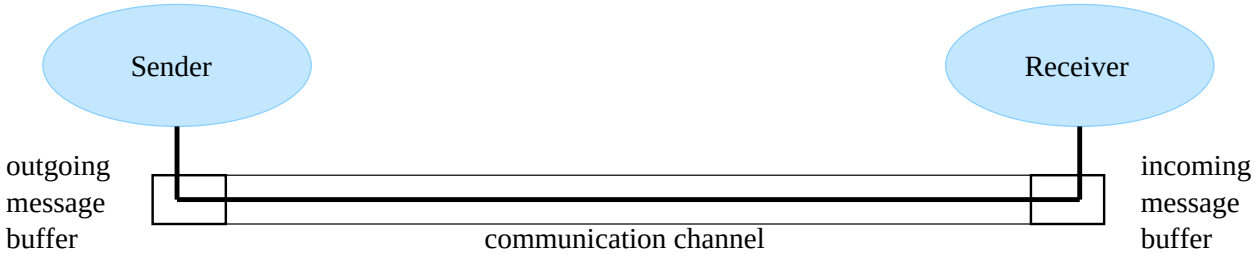
(Lecture 12: Failures and Fault Tolerance,
13th November 2006)

Omission Failures

When a process or channel fails to do something that it is expected to it is termed an **omission failure**.

Process omission failures: A process makes an omission failure when it **crashes** — it is assumed that a crashed process will make no further progress on its program. A crash is considered to be **clean** if the process either functions correctly or has halted. A crash is termed a **fail-stop** if other processes can detect with certainty that the process has crashed.

Communication omission failures: Communication omission failures may occur in the sending process (*send-omission failures*), the receiving process (*receive-omission failures*) or the channel (*channel omission failures*).



Arbitrary Failures

- The term "**arbitrary**" or "**byzantine** failure" is used to refer to the type of failure in which any error may occur.
- In a process, arbitrary behaviour may include setting incorrect data values, returning a value of incorrect type, stopping or taking incorrect steps.
- In a channel, arbitrary behaviour may include duplication or corruption of messages.
- Most protocols include mechanisms to overcome arbitrary failures in a channel. For example, checksums to detect corruption and sequence numbers to detect duplication.
- Arbitrary failures in a process are less easy to detect and can have a profound impact on a system in which several processes need to cooperate.
- For example, consider the behaviour of the leader election algorithms if one of the processes behaved erratically and did not follow the protocol of the algorithm.

Synchronous Systems and Timing Failures

Recall that a **synchronous** distributed system is one in which each of the following are defined:

- upper and lower bounds on the time to execute each step of a process;
- a bound on the transmission time of each message over a channel;
- a bound on the drift rate of the local clock of each process.

If one of these bounds is not satisfied in a synchronous system then a **timing failure** is said to have occurred.

Few real systems are synchronous (they can be constructed if there is guaranteed access to resources) but they provide a useful model for reasoning about algorithms — **timeouts** can be used to detect failed processes. In an asynchronous system a timeout can only indicate that a process is not responding.

Fault Tolerance

Unfortunately it is not possible to avoid failures. Instead we must aim to build systems in which we minimize the impact of failures when they do occur. Some mechanisms which can be used include:

Atomicity and Rollback A **transaction** is a sequence of computational steps which are treated as atomic. In transaction processing, mechanisms are included to allow **rollback** if the complete transaction cannot be committed due to failure or error.

Replication Having several copies of a service available within a system means that failure of any one server may be hidden from an application.

Persistence Process crash may also be masked by a persistent service, such as that provided in CORBA, in which data stored on disk may be used to restore the state of a server which is automatically restarted.

Transactions

A transaction

- runs on shared entities such as databases, files or remote objects;
- is a unit of consistency, meaning that it transforms the entity from one consistent state to another consistent state; and
- has effects which are either made permanent (**committed**) or cancelled (**aborted**) so that the entity reverts back to its original state.
- To cope with failures and concurrency issues transaction systems must guarantee the **ACID** properties:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

ACID Implications — Recoverability

The requirements of atomicity and durability mean that entities must be **recoverable**.

An entity is **recoverable** if when the server process crashes (due to hardware fault or software error) the changes due to completed transactions are not lost. Furthermore when the server is replaced by a new server (or, equivalently, restarted) the entity can be restored.

This implies that the state of each entity must be recorded in permanent storage, for example, written to disk. Moreover this must be done by the time that the server acknowledges the completion of a client's transaction.

Transaction Coordinator

Each transaction is created and managed by a coordinator which should implement the following operations (client-side):

`openTransaction()` starts a new transaction and returns a unique **transaction ID (TID)** `trans`. This identifier is used in other operations in the transaction.

`closeTransaction(trans)` ends a transaction; the return value is used to indicate the success or failure. Instigates the saving of all recoverable objects accessed by the transaction. A return value of `commit` indicates that the transaction has been committed; a return value of `abort` indicate that it has been aborted.

`abortTransaction(trans)` is used when the client wants to abort the transaction. All recoverable objects accessed by the transaction will be **rolled back** to their state before the transaction started.

All the client's operations between the `openTransaction()` and the `closeTransaction(trans)`/ `abortTransaction(trans)` calls are operations of the transaction.

Server Process Crashes

A transaction can also be aborted by the server process as the result of a process crash. We assume in this situation that the server will be restored or replaced after an unexpected failure.

- The new (restored) server process aborts any uncommitted transactions which were in progress when the crash occurred. It also starts a recovery procedure which reverts all objects to the values produced by the most recently committed transaction.
- The client will be made aware of the server crash when an operation returns an exception after a timeout. If the server is restored while the transaction is still in progress the client is again informed via an exception as the transaction will no longer be valid.

Servers can assign expiry times to transactions in order to detect client crashes. Any transaction which has not committed by its expiry time will be aborted by the server.

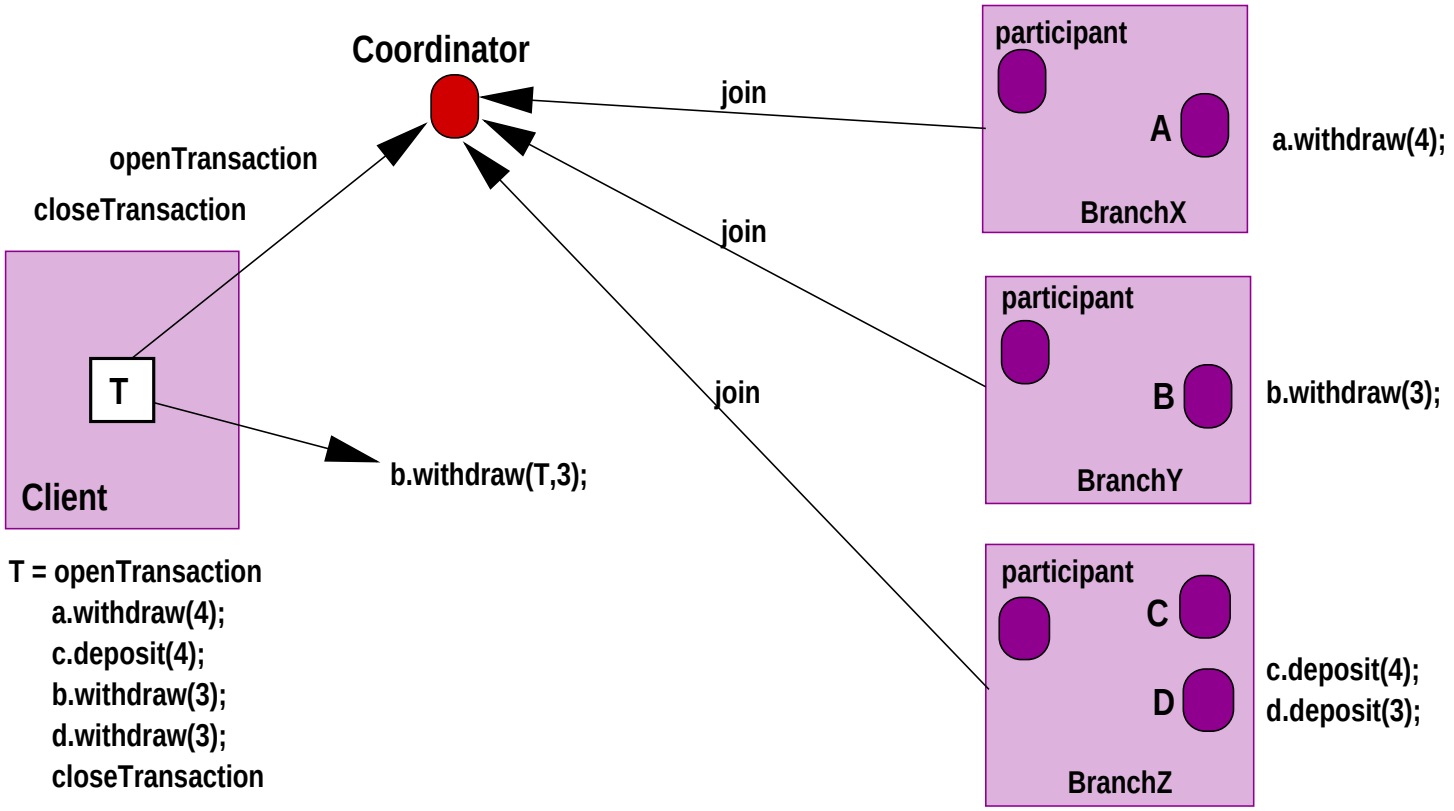
Distributed Transactions

- A transaction is **distributed** if it invokes operations in several different servers.
- One of the servers takes on the role of **coordinator** to ensure that the atomicity property is kept and that either all the servers commit or they all abort.
- This process is managed by a **commit protocol**, the most common of which is the **two-phase commit protocol**.
- Each server applies concurrency control locally to its own objects to ensure that the local effects are serially equivalent.
- However additional procedures are needed to ensure that distributed transactions are globally serialised.
- To enhance concurrency it is often useful to organise transactions in terms of **nested transactions**: transactions within transactions, with subtransactions able to proceed in parallel.

Role of Coordinator

- A client starts a transaction by sending an `openTransaction` request to a coordinator in any server.
- The coordinator returns the transaction ID which now contains a server identifier (IP address) in addition to the transaction number assigned by the server.
- The coordinator will keep the role of coordinator with respect to that transaction throughout its lifetime, and in the end will be responsible for committing or aborting it.
- Other servers which manage objects accessed by the transaction are termed **participants** and each must be involved in the decision to commit or abort. Each has a reference to the coordinator.
- During the course of the transaction the coordinator records a list of references to the participants that have been involved, via the `join` method that each participant invokes on the coordinator whenever it becomes involved in a transaction.
- Each participant must manage its own recoverable objects.

Coordinator Example



Although separate for clarity here, the coordinator would be situated in one of the servers involved in the transaction (`BranchX` say).

Two-phase Commit Protocol

- When an distributed transaction comes to an end either all its operations are carried out or none of them (atomicity).
- Since events at each participant (which will not necessarily be known at the coordinator) may mean that a transaction has to be aborted, the commitment of a transaction proceeds in two steps.
- Thus each server can abort its part of the transaction due to local conditions, and then force the whole transaction to be aborted.
- In the first phase of the protocol each participant votes for the transaction to be committed or aborted.
- Each participant is allowed only one vote, so a vote to commit is binding even in the event of a subsequent failure at the server.
- A participant is said to be in a **prepared state** for a transaction if it will eventually be able to commit it.
- In the second phase of the protocol every participant in the transaction carries out the joint decision.

Two-phase Commit Protocol (2)

Phase 1

1. Coordinator sends a `canCommit?` request to each participant.
2. When a participant receives a `canCommit?` request it replies with its vote (`Yes` or `No`). If voting `Yes` it prepares to commit by saving objects in permanent storage. If voting `No` it aborts immediately.

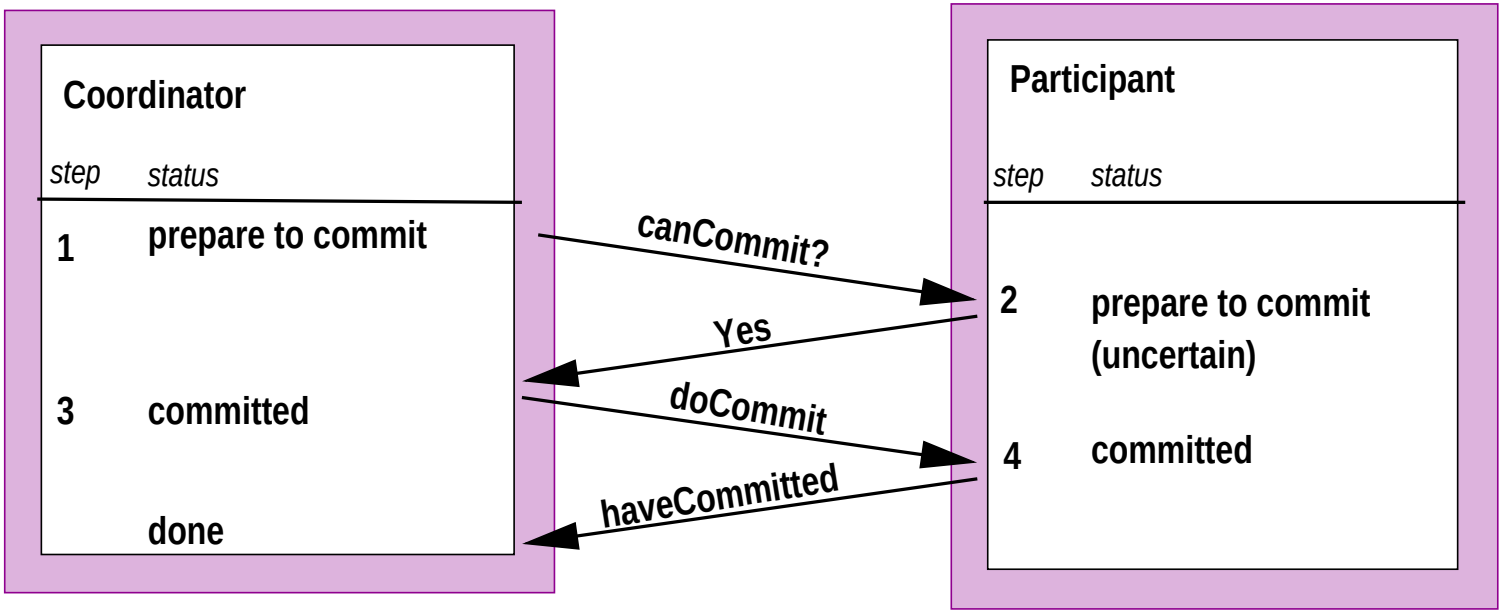
Phase 2

3 The coordinator collects votes (including its own).

- If there are no failures and all votes are `Yes` coordinator decides to commit and sends everyone a `doCommit` request.
- Otherwise the coordinator decides to abort and sends `doAbort` request to each participant that voted `Yes`.

4 Participants that voted `Yes` are waiting for a message from the coordinator. When a participant receives a `doCommit` or `doAbort` request it acts accordingly and in the case of commit sends a `haveCommitted` message to the coordinator.

Two-phase Commit Protocol (3)



Without errors this is quite straightforward but it must also work correctly when some of the servers fail, messages are lost or servers are temporarily unable to communicate. A participant can use the `getDecision` method of the coordinator if it is left at step 2.

Transactions in CORBA

- In CORBA transactions are supported by the *Object Transaction Service*.
- IDL interfaces allow client transactions to include not just multiple accesses to a single object but also multiple objects at multiple servers.
- The client is provided with operations to specify the beginning and end of a transaction, as explained on the earlier slide.
- The TID is implicitly passed as an additional argument in each included remote method invocation, i.e. those made between calls to start the transaction and those to complete/abort it.
- The client ORB maintains a context for each transaction, which it propagates with each remote method invocation in that transaction.
- CORBA objects can be made transactional by making their interfaces extend an interface called `TransactionalObject`.

Concensus Algorithms

- In a general concensus problem we assume that there are N processes $P_i, i = 1, \ldots, N$, each of which starts in an **undecided** state.
- As a group they must agree on some value V. Each process **proposes** a single value v_i.
- The processes communicate, sending their values to each other.
- Each process then sets the value of a **decision variable** d_i.
- Once d_i has been set, process P_i enters the **decided** state and cannot change d_i.
- In a failure-free system, the decision can be made on the basis of some previously decided function, for example $\mathsf{majority}(v_1, \ldots, v_n)$.
- In an asynchronous system in which omission failures can occur the algorithm may not terminate.
- In a system in which arbitrary failures can occur, a failed process may propagate incorrect values.

Requirements of Concensus Algorithms

Every run of a concensus algorithm should satisfy the following three properties:

Termination: Eventually every correct process sets its decision variable.

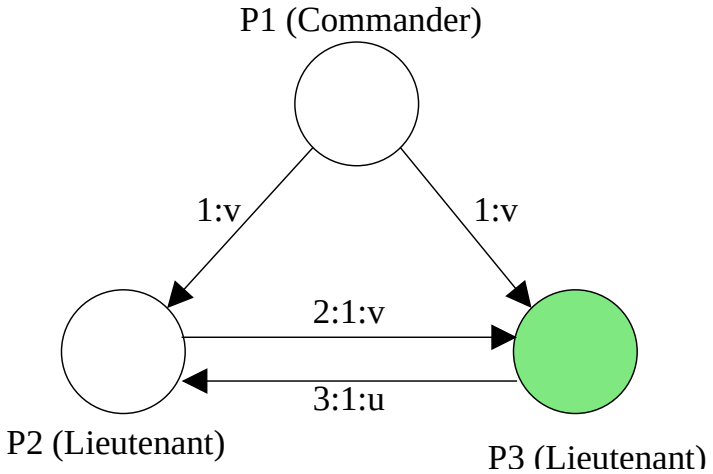
Agreement: All correct processes set their decision variables to the same value, that is, if P_i and P_j are both correct and both have entered the **decided** state, then $d_i = d_j, (i, j, = 1, 2, \ldots, N)$.

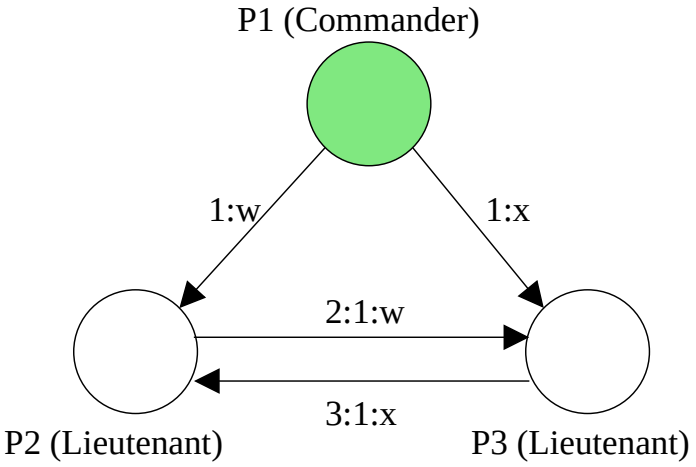
Integrity: If the correct processes all proposed the same value, then any correct process in the **decided** state has chosen that value.

Different forms of integrity are sometimes used for variations of the problem. For example, a weaker form states that the decision value chosen should be one proposed by one of the correct processes.

Byzantine Generals Problem

The byzantine generals problem is a variation on the consensus problem. We assume that there is a set of generals, who have to collectively agree to *attack* or *retreat*. One general is the commander; the remainder are the lieutenants. The commander issues an order, which the lieutenants then discuss amongst themselves by exchanging messages, reporting the instruction they have been given. A "failed" general is treacherous and may send erroneous messages.





Interactive Consistency Problem

The interactive consistency problem is another variation on the consensus problem. Here we assume that each process proposes a value to be included in a result vector, the **decision vector**. The goal of the algorithm is for the correct processes to agree on this vector. For example, this might be used in order for a set of processes to obtain a consistent set of information about their respective states.

It can be shown that the consensus problem, the byzantine generals problem and the interactive consistency problem are all equivalent in the sense that if we can find a solution for one of them we can apply the solution to all of them (with suitable modification).

It can also be shown that solving the consensus problem is equivalent to solving reliable and totally ordered multicast.

Consensus in a Synchronous System ($\leq f$ faults)

```
Algorithm for process Pi in G; algorithm proceeds in f+1 rounds

On initialization
Values(1,i):= {vi}; Values(0,i) = {};

In round r ( 1 <= r <= f+1 )
   B-multicast(G, Values(r,i) - Values(r-1,i));
                       // Send only values that have not been sent
   Values(r+1,i):= Values(r,i) ;
   while (in round r)
     {
         On B-deliver(vj) from some Pj
         Values(r+1,i) := Values(r+1,i) + vj;
     }
After (f+1) rounds
Assign di = minimum(Values(f+1,i));
```

Byzantine Generals in a Synchronous System

- It has been shown that it is impossible to solve the byzantine generals problem (and the other consensus problems) in an asynchronous system.
- In a synchronous system a limit must be put on the number of processes which may fail.
- It is impossible to solve the byzantine generals problem if too large a proportion of the processes are allowed to be faulty: $N \leq 3f$.
- In particular, in the case of three processes and one faulty, the problem has no solution.
- This can be improved upon if processes sign their messages.
- Even in this case the algorithm is costly in terms of the number of messages sent and only justified when the threat is significant.
- For unsigned messages, an algorithm needs $f + 1$ rounds where f is the number of faults which may occur.

Solution with one faulty process

The correct generals reach agreement in two rounds of messages:

1. In the first round, the commander sends a value to each of the lieutenants.
2. In the second round, each of the lieutenants send the value it receives to its peers.

When both rounds are finished the correct lieutenants simply apply a majority function to determine the value

- If the commander is faulty, all the lieutenants faithfully report the values they received but there will be no agreement (distinguished symbol $\perp$).
- If one of the lieutenants is faulty, the correct lieutenants will receive $N-2$ replicas of the correct value, plus one incorrect one: the majority function will determine the correct value.