COMP28112 Lecture 12

Fault Tolerance - Transactions

Key Definitions

- *"A characteristic feature of distributed systems that distinguishes them from single-machine (centralized) systems is the notion of **partial failure**".* [Tanenbaum, p.321]
- The goal is to **tolerate faults**, that is, to operate in an acceptable way, when a (partial) failure occurs.
- Being fault tolerant is strongly related to **dependability**:
  - *"Dependability is defined as the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers" [IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance, http://www.dependability.org]*

Requirements for Dependability

- **Availability**: the probability that the system operates correctly at any given moment.
- **Reliability**: length of time that it can run continuously without failure.
- **Safety**: if and when failures occur, the consequences are not catastrophic for the system.
- **Maintainability**: how easily a failed system can be repaired.

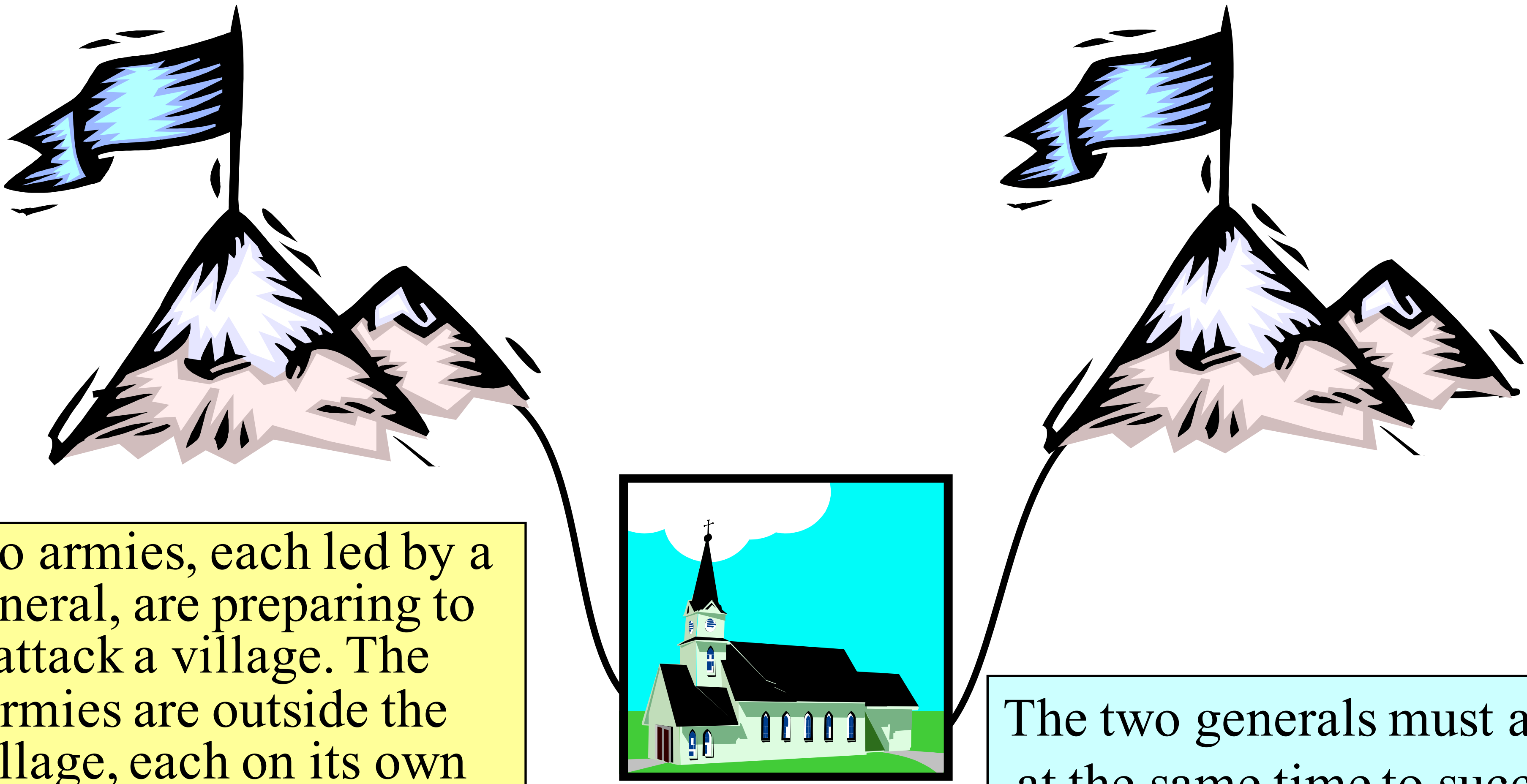
Types of Failures

- **Crash:**
  - Server halts!
- **Omission failures:**
  - Server fails to respond to incoming requests
  - Server fails to receive incoming messages
  - Server fails to send messages
- **Response failures:**
  - A server's response is incorrect
- **Timing failures**:
  - Server fails to respond within a certain time
- **Arbitrary (byzantine) failures:**
  - A component may produce output it should never have produced (which may not be detected as incorrect) – arbitrary responses at arbitrary times.

Benign (i.e., omission/timing) failures are by far the most common; we'll see problems related to byzantine failures later on.

The two generals' problem (or paradox)…

(pitfalls and challenges of communication with unreliable links…)



Two armies, each led by a general, are preparing to attack a village. The armies are outside the village, each on its own hill. The generals can communicate only by sending messengers passing through the valley.

The two generals must attack at the same time to succeed!

http://en.wikipedia.org/wiki/Two_Generals'_Problem

Failure masking using redundancy

- **Physical redundancy**:
  - A well-known engineering technique (to give an extreme example, B747s or A380s have four engines but – subject to certain conditions – can fly on three)
  - Even nature does it!
- **Time redundancy**:
  - An action is performed, if need be, again and again.
  - Especially helpful when faults are transient and intermittent.
- **Information redundancy**:
  - e.g., send extra bits when transmitting information to allow recovery.

Redundancy…

- …creates several problems:
  - Consistency of replicas (e.g., all data need to be updated).
  - Should improve (overall) system performance.

  (we'll return to these!)
- …costs money!

But, above all:

We still need to make sure that any failure won't leave our system in an inconsistent (corrupted) state!

Example: A Simple Application (a client communicating with a remote server)

Transfer £100 from account 1 to account 2

- x = read_balance(1);
- y = read_balance(2);
- write_balance(1, x – 100);
- write_balance(2, y + 100);

Crashes can occur at any time during the execution
What problems can arise because of this?

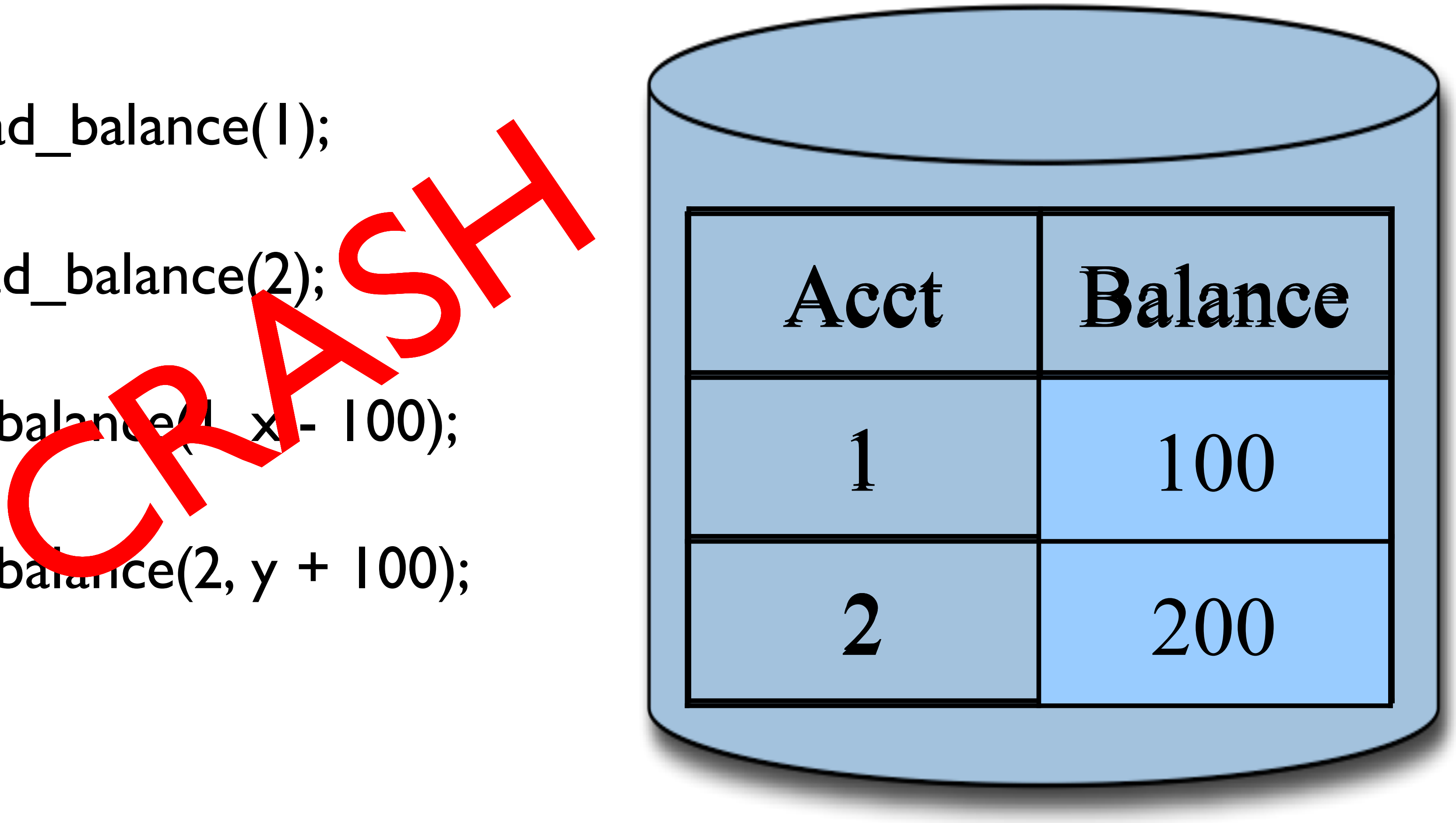
Crash

```
x = read_balance(1);

y = read_balance(2);

write_balance(1, x - 100);

write_balance(2, y + 100);
```



Acct	Balance
1	100
2	200

All-or-Nothing

- Either ALL operations execute or NONE
  - x = read_balance(1);
  - y = read_balance(2);
  - write_balance(1, x - 100);
  - write_balance(2, y + 100);

The sequence of operations MUST execute as an **ATOMIC** operation

Multiple users can be transferring funds simultaneously. What problems can arise because of this?

Concurrent Users

Transfer £100 from acct 1 to 2	Transfer £300 from acct 1 to 2
x = read_bal(1)	u = read_bal(1)
y = read_bal(2)	v = read_bal(2)
write_bal(1, x-100)	write_bal(1, u-300)
write_bal(2, y+100)	write_bal(2, v+300)

Possible Sequence of Events

1	x = read_bal(1)
2	u = read_bal(1)
3	v = read_bal(2)
4	write_bal(1, u-300)
5	y = read_bal(2)
6	write_bal(1, x-100)
7	write_bal(2, y+100)
8	write_bal(2, v+300)

Acct	Balance
1	0
2	500

What you expect

Acct	Balance
1	-300
2	600

What you got

Acct	Balance
1	0
2	500

The two transfers got in each other's way

Does all this remind you anything?

Isolated Execution

- We must ensure that “concurrent” applications do not interfere with each other
  - But what does interfere mean?

Serial (=Sequential) Executions

- Concurrent executions do not interfere with each other if their execution is equivalent to a serial one:
  - The reads and writes get the same result as if the transfers happened one at a time (i.e. they don’t interleave).
- Simple but naive solution:
  - One transfer at a time
  - Not scalable and very very slow
- How do we maximise concurrency without corrupting the data?
  - Good question!

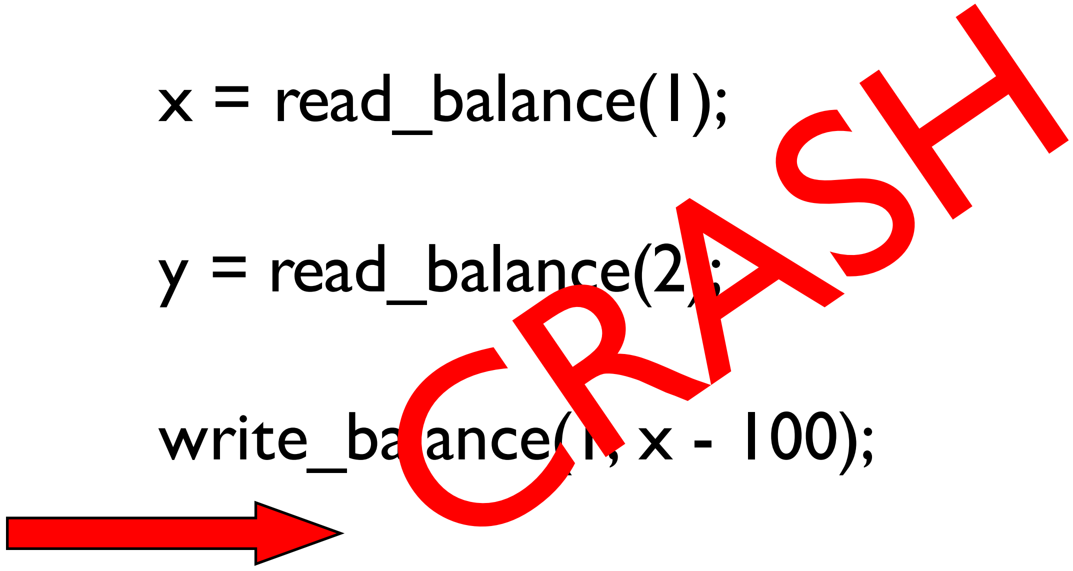
Can crashes cause problems?

```
x = read_balance(1);

y = read_balance(2);

write_balance(1, x - 100);

write_balance(2, y + 100);
```



Acct	Balance
1	0
2	300

Data surviving crashes could be in anyone of these three states

Acct	Balance
1	100
2	200

Acct	Balance
1	0
2	200

Acct	Balance
1	0
2	300

Durable

- Updates are persistent once the application successfully completes

Acct	Balance
1	0
2	300

An application should not violate a database's integrity constraints

- Balance of ALL customers should not exceed their overdraft limit
- All account holders have a name and an address

- Transfer £500 from account 1 to account 2
  - Transfer should not be permitted if overdraft limit is £200 for account 1

Acct	Balance
1	100
2	200

Consistency

Wouldn’t it be great if we had an abstraction (and an implementation) that provided us with the ACID properties?

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**

Transactions (=individual, indivisible operations) to the rescue

- Originated from the database community
- Simple way to write database applications
  - Provides the ACID properties
  - Transaction either commits or aborts
- Fast, recovers from all sorts of failures, highly available, manages concurrency, ...
- In use everywhere and everyday

```
begin_tx
   ...
   ...
   ...
commit_tx
```

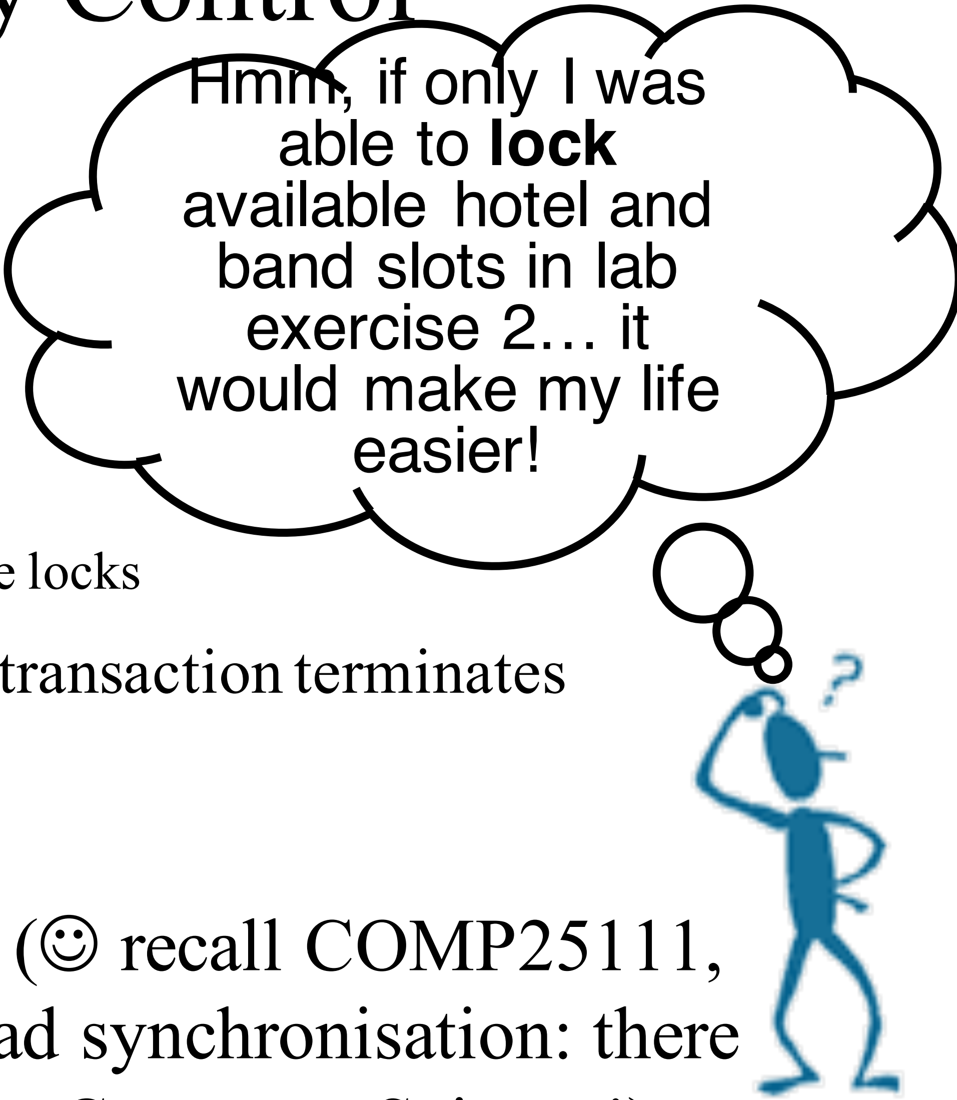
How Transactions are Implemented

- Managing multiple “simultaneous” users
  - **Concurrency control** algorithms
  - Ensure the execution is equivalent to a “serial” execution (key assumption: transactions have a short duration in the order of milliseconds: you don’t want to “block” other transactions for too long)

- Durability
  - **Recovery algorithms**
  - Replay the actions of committed transactions and undo the effects left behind by aborted transactions

Concurrency Control

- Two-phase locking
  - "**Acquire locks**" phase
    - Get a read lock before reading
    - Get a write lock before writing
    - Read locks conflict with write locks
    - Write locks conflict with read and write locks
  - "**Release locks**" phase when the transaction terminates (commit or abort)


Hmm, if only I was able to **lock** available hotel and band slots in lab exercise 2… it would make my life easier!

What does all this remind you of? (☺ recall COMP25111, lectures on semaphores and thread synchronisation: there are some key problems in core Computer Science!)

Conclusion

- Redundancy is the key to deal with failures
- We need to avoid corruption of data due to failures:
  - Use transactions.

- Reading:
  - Tanenbaum *et al*: Sections 1.3.2, 8.1-8.3 (weak on transactions).
  - Coulouris *et al (4th ed)*: Sections 2.3.2, 13.1, 13.2.
  - Coulouris *et al (5th ed)*: Sections 2.4.2, 16.1, 16.2.