Distributed Systems
Fault Tolerance

timofei.istomin@unitn.it

credits for the slides to:
Giuliano Mega
Luca Mottola
Davide Frey

Outline

- Introduction
- An example: client-server communication
- Reliable group communication
- Agreement in process groups
- Atomic commitment

Why be fault tolerant

- Tolerate faults — operate correctly (without failing) despite faults
- Why is this important?
- Your laptop:
  - Pr[CPU fault in next 30 days] 0.5% [Nightingale 2012]
  - that's a 1.64 years expectation for failure
- If you're Google, however…
  - ~ 1 million servers
  - ~ 5.000 faults/month
  - ~ 166 faults/day
  - ~ 7/hour

Why be fault tolerant

- In August 2013, Google had 1~5 minutes of outage
  - Internet traffic dropped by 40%
  - estimated USD 500,000 revenue loss
- Google cannot afford downtime

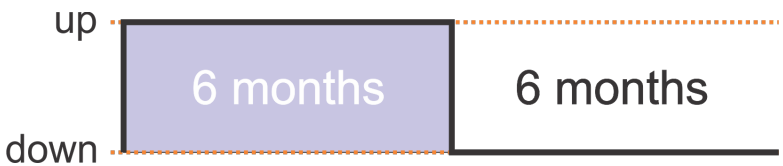
- Distributed system mentality:
  - faults are a reality, plan for them

Metrics of Dependability

"Five nines" availability 99.999%; ~5min downtime per year

1. Availability
   - pick a random instant t in time;
   - what's the probability that the system is "up"?
2. Reliability
   - system was brought up at time t_0;
   - what's the probability it's still running by $t + t_0$?

Higher = more dependable. Can a system be reliable but not available?



Metrics of Dependability

3. Safety
   - probability that the system will not suffer a catastrophic failure by time t
   - e.g. a circuit burns in a life support system. What's the probability somebody ends up without oxygen?
   - The lower, the safer.
4. Maintainability:
   - system goes down at time t_0
   - what's the probability it's restored by $t + t_0$?
   - Higher = more maintainable
   - correlates to availability
5. Security*: ability to prevent unauthorized access/handling. Not covered here.

Absolute dependability?

Do absolutely dependable systems exist?

Of course not!

Applying the theory

- The Fault Tolerance theory provides common terms to *reason* about faults and a set of *techniques* and *algorithms* to make a distributed system tolerate the faults
- The algorithms are proven to hold certain properties under certain conditions (e.g. types and number of simultaneous faults)
- Should unexpected faults occur – no guarantees are given, the system will probably fail

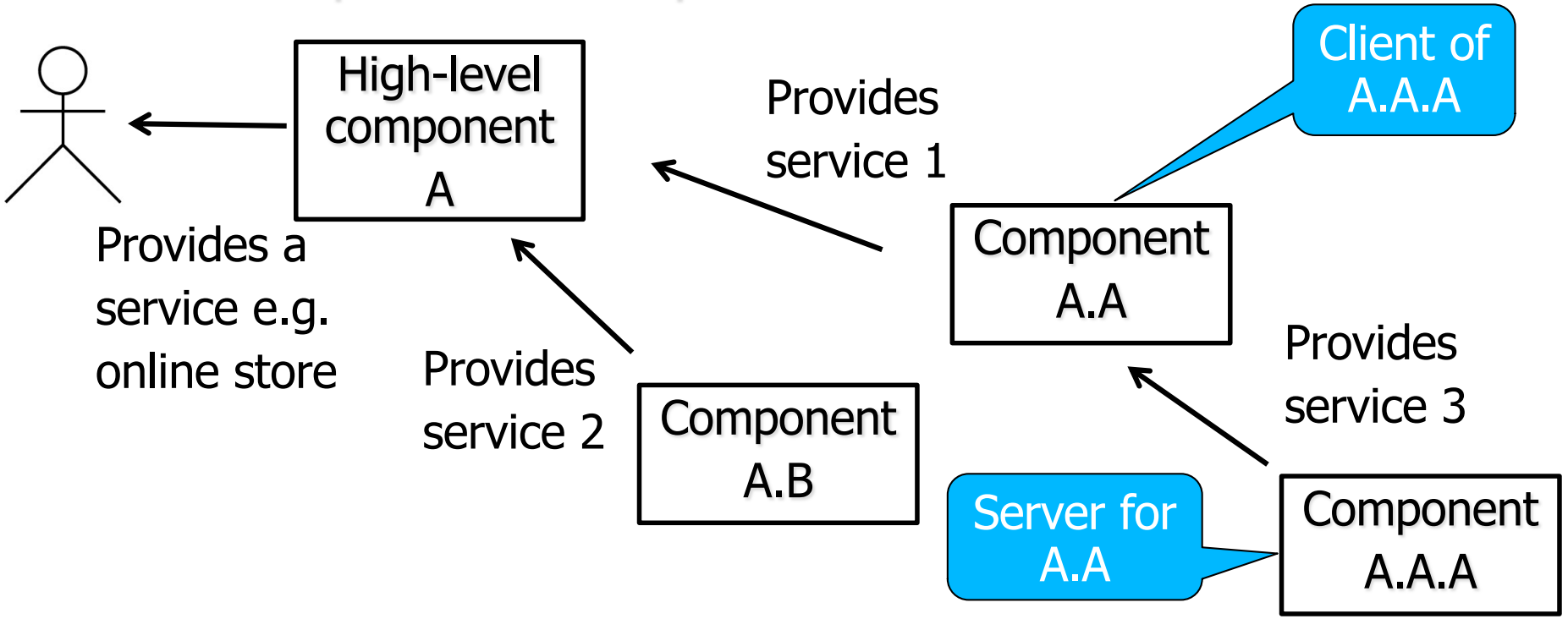
Applying the theory

- There are many types of systems with different requirements (reliability, cost, performance, etc.)
- It is up to the system designer to decide which techniques are appropriate for the system
  - Which faults are likely to occur
  - How critical are those faults
  - How much resources we can spend to tolerate them

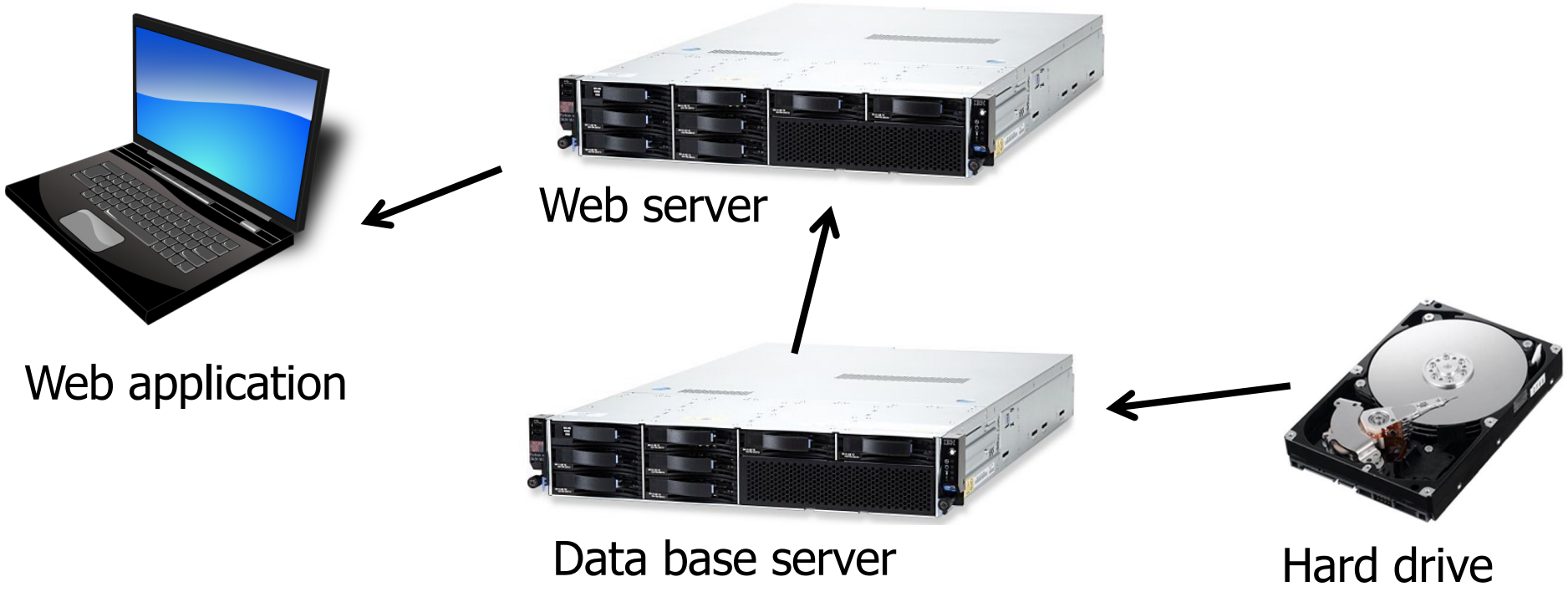
Dependency hierarchy

Any system consists of components

- Components provide *services*
- Higher-level components *use* services of lower-level components to implement their own service



Dependency hierarchy



Many other components are not shown:

- OS, software libraries, network stack, wires, CPU, memory, etc...

Services, servers, clients

- *Service* – a set of operations that can be requested from the *server* by its *clients*
- Service specification:
  - Request syntax (formats)
  - Operation semantics:
    - What should the server do
    - What is the correct reply
    - How should the server's internal state change
  - What is the timing of the reply
  - What faults are likely to occur

Server's promise

Faults and Failures

- A system fails when it cannot provide its service correctly

- A failure of a component (subsystem) can be seen as a fault at the system level, e.g.:
  - hard drive failure
  - packet loss in the network
  - bit flip in memory (e.g. if an alpha particle hits the memory module)
  - software bug (e.g. a deadlock or race condition)
  - …

Faults and Failures

- Dilemma: improve reliability of the components or implement error-handling mechanisms at the higher level?
  - E.g. to deal with memory faults:
    - shield the memory module or
    - introduce redundancy: error-detecting coding or replication
- Both are usually done (to some extent)
- Often it is more cost-efficient to use cheaper components but tolerate their failures

Tolerating faults

- Building a dependable system demands the ability to understand, and deal with faults
- This requires understanding of *how* components might fail
- Classifying failures help: *failure modes*

Server Failure Modes

Type of failure	**Description**
Omission failure - *Receive omission* - *Send omission*	The server fails to respond to incoming requests - it fails to receive incoming messages - it fails to send messages
Timing (performance) failure	The server's response lies outside the specified time interval
Response failure - *Value failure* - *State transition failure*	The server's behaviour is wrong: - the value of the response is wrong - the server deviates from the correct flow of control
Crash failure	The server halts, but is working correctly until it halts
Arbitrary (aka Byzantine) failure	The server may produce arbitrary responses at arbitrary times

Crashed or just slow?

- If there's no reply from the server
  - It might have crashed or
  - It might exhibit a timing failure
    - overloaded with requests, or network delay
- How the client can tell one from the other?
  - It depends on how the system is built: is it *synchronous* or not?

[A]synchronous systems

The system is synchronous if

- communication delays
- execution time of every operation
- and clock drift of the components

are all bounded

In other words, the network and the processes cannot demonstrate timing (performance) failures by design

Most distributed systems are *asynchronous*

Failure detectors

- In a synchronous system (with reliable channels) it is easy to detect a crash
  - just ping the server periodically: if no reply within specified time — it has crashed
- In an asynchronous system it is not possible in general to tell a crashed server from a slow one
  - there are no reliable *failure detectors* in asynchronous systems: a process might be *suspected* to have crashed
- Failure detector – an algorithm (possibly distributed) that tries to detect if a server has crashed

Which failure mode?

- Fault tolerance strategies rely, in general, on being able to tell when and how something can fail
- But it is **not** easy to tell which kind of failure is going on! E.g.:
  - server crashed vs. lost request
  - lost request vs. lost reply
  - all of the above vs. slow server
  - even worse in the case of arbitrary (Byzantine) failures: correct vs. malicious vs. crazy behaviour
- Bottom line: fault tolerance is not easy!

Which failure mode

- Anecdote: Amazon's Simple Storage Service (S3)
  - cloud-based data storage
  - applications that use S3 as backend: Dropbox, Instagram, Spotify, Foursquare, etc.
- August 2008: outage for an entire day
  - datacenters: secure environment
  - was well-equipped to deal with crash failures
  - sufficient replication and redundancy
  - bit flip on a memory module – *which failure mode?*
  - propagated by means of a gossip protocol
  - redundancy didn't help: replicas were corrupt

Faults and Failures

- Faults can also be classified according to the frequency at which they occur
  - Transient faults occur once and disappear
  - Intermittent faults appear and vanish on their own accord
  - Permanent faults continue to exist until the failed components are repaired

Dealing with faults

- Hide completely *(mask)*
  - Triple hard drive redundancy: two right values outvote the wrong one (mask the value error)
  - Process replication: if one crashes, we can redirect the request to another one (mask the crash)
  - Channel redundancy: one link is down, use the other one (mask the omission)
- Change the type to less severe *(convert)*
  - drop a packet if CRC is wrong: value -> omission
  - use acknowledgements and retries: omission -> timing
  - CPU replication: just crash if two CPUs produce different results for the same operation: byzantine -> crash

Outline

- Introduction
- An example: client-server communication
- Reliable group communication
- Agreement in process groups
- Atomic commitment

Reliable client-server communication

- We'll have a look at how difficult it can be to provide failure masking in general in a simple client-server system
- complete masking of failures is in general just not feasible
- becomes critical in systems that try to achieve network transparency, such as RPCs

```
{  ...
        result = doOperation();
}
```

Reliable client-server communication (2)

- RPC was supposed to unify local and distributed programming models
  - call to local function the same as call to remote function: transparency is of the essence
- But, in reality a number of problems arise:
  - Server cannot be located
  - Lost requests
  - Lost replies
  - Server crashes
  - Client crashes
- Ever wondered if the `+` operator in a C program would work?

The benign cases

- If the server cannot be located by the client, the client can handle this as an exception
  - Annoying but tolerable
  - Not really masking much
    - yet, not much that can be done
- But what if the request or the reply are lost (or the server crashed?)
  - Did it complete the requested operation?
- Even if the result was received…
  - How many times the server executed the requested operation?

At-least-once semantics

- If the server answer does not arrive in time, the RPC system can try to mask the fault by resending the request
- In case all the requests are lost, the server does not execute the function
  - An exception is eventually generated
- If some of the replies are lost, the server may have executed the function once or more times

Idempotent operations

- Some requests are safe to complete many times, i.e., they are idempotent
  - e.g., set a parameter value, or read a value from storage
- … but others are not
  - e.g., bank transfer
- If all operations provided by the server are designed as idempotent, it is safe to use a RPC system with the *at-least-once* semantics

At-most-once semantics

- Sequence numbers are added to the requests
- A RPC client may retransmit the same request
- The RPC server keeps a history of executed requests (based on the sequence number)
  - it executes the function the first time
  - stores the result in the history of requests
  - if a duplicate request arrives (same seqnum), just sends back the result from the history

At-most-once semantics

- How long should the server keep the history?
  - Until the next sequence number arrives from the client
  - If none arrives, clean up after a reasonable timeout
- Though the client might be just very slow
  - Notify the client that the session ID it uses has expired and it has to initialize a new session
  - At least it gets informed that something went wrong and can do some checks to figure out whether the operation was executed or not

RPC invocation semantics

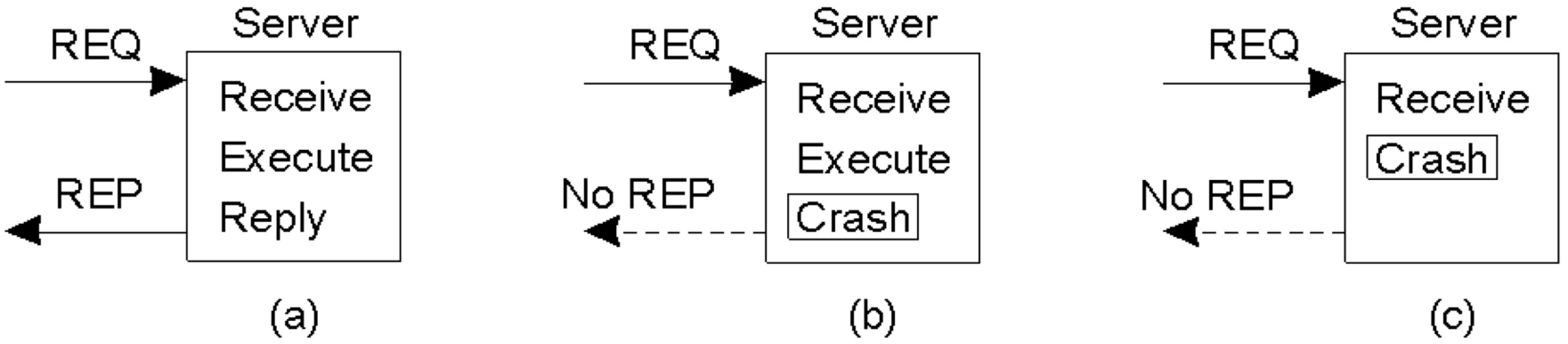
- So, when you call a remote function you might either get the result or an exception (e.g., timeout)

Semantics	**Got a result**	**No result (exception)**
At least once	executed one or more times	not executed or executed one or more times
At most once	executed once	not executed or executed once

- Both are different from the *exactly once* semantics of a local procedure call
- Different systems provide different guarantees:
  - Sun RPC Java uses at least once, Java RMI at most once
- There's no network transparency: the program that uses remote calls should know that they are remote

The server crashes

- The server might crash at any time and reboot with partial amnesia
- The problem is that the client cannot know when the server crashed



Example: a print server

- The server performs two operations:
  - Print the requested document (P)
  - Send a confirmation message (M)
- We want to explore all possibilities, so the server might either
  - Print, then Send confirmation or Send confirmation, then Print
- The server can crash (C) at any time
  - Before beginning: C (-> P -> M) or C (->M -> P)
  - At the end: P -> M -> C or M -> P -> C
  - In the middle: P->C (->M) or M-> C (->P)

Client behaviour

- Assume the client knows the server crashed, but not *when* and doing *what*
- Suppose that the server later recovers
- The RPC runtime at the client wants to mask the fault completely. Four possibilities:
  - Always reissue request
  - Reissue only if confirmation was not received
  - Never reissue request
  - Reissue only if confirmation was received

Print server and client

Client	Server					
	Strategy P -> M			Strategy M -> P		
Reissue strategy	PMC	PC(M)	C(PM)	MPC	MC(P)	C(MP)
Always	DUP	DUP	OK	DUP	OK	OK
Never	OK	OK	ZERO	OK	ZERO	ZERO
Only if ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only if not ACKed	OK	DUP	OK	OK	ZERO	OK

- Take away messages:
  - Some strategies may be better than the others but none can provide exactly-once semantics
  - Masking failures is difficult: one-size-fits-all is in general not possible
  - Meaningful failure recovery is normally application-specific

Redundancy

- The key technique to tolerate and mask failures is redundancy
  - Information redundancy
    - e.g., using Hamming codes
  - Time redundancy
    - Hard luck? Try again!
  - Physical redundancy
    - Two is better than one
    - First implemented in biological systems

Group failure masking

Physical redundancy can be used to mask the presence of faulty components

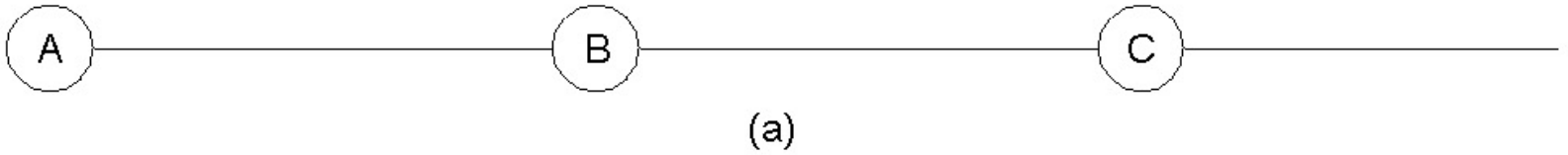
- The work that should be done by a component is taken care of by a group of identical components (replicas)
- Non-faulty components continue to work when some of the others fail
- The group output is a function of the outputs of its members, e.g.:
  - the reply of the fastest component (masks crashes and delays)
  - the majority vote (masks value failures)

Group failure masking

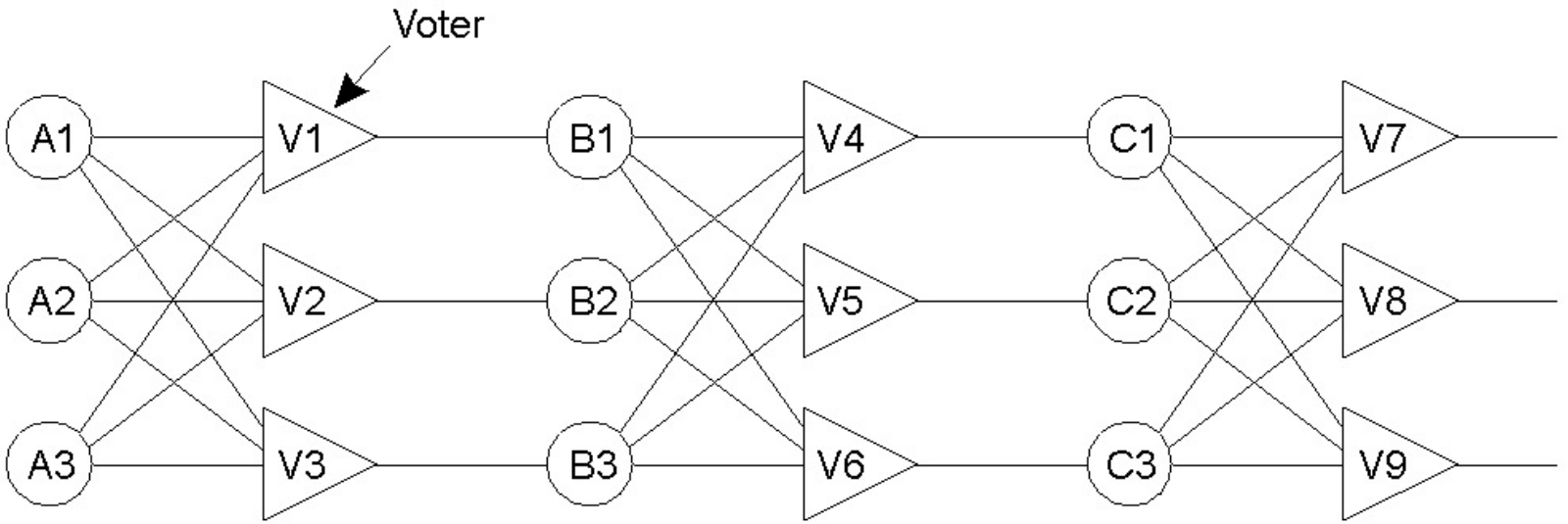
- How many replicas do we need?
  - If k replicas fail silently, then k+1 servers allow the system to be k-fault-tolerant
  - If failures are Byzantine, matters become worse: 2k+1 replicas are the required minimum to achieve k-fault tolerance (with majority voting)
    - Still provides a majority of k+1 correct processes, even if k behave erratically…
- We cannot be sure that no more than k servers will ever fail simultaneously, but we can use a "reasonable" value

Triple modular redundancy

Group masking technique applicable for electronic circuits



(a)

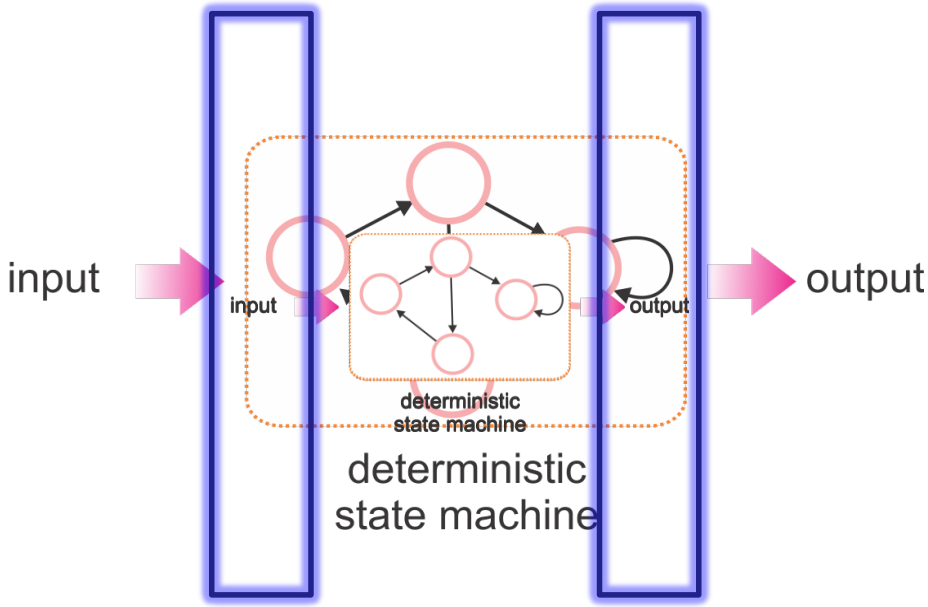


(b)

Can we do the same in a distributed system?

Process resilience

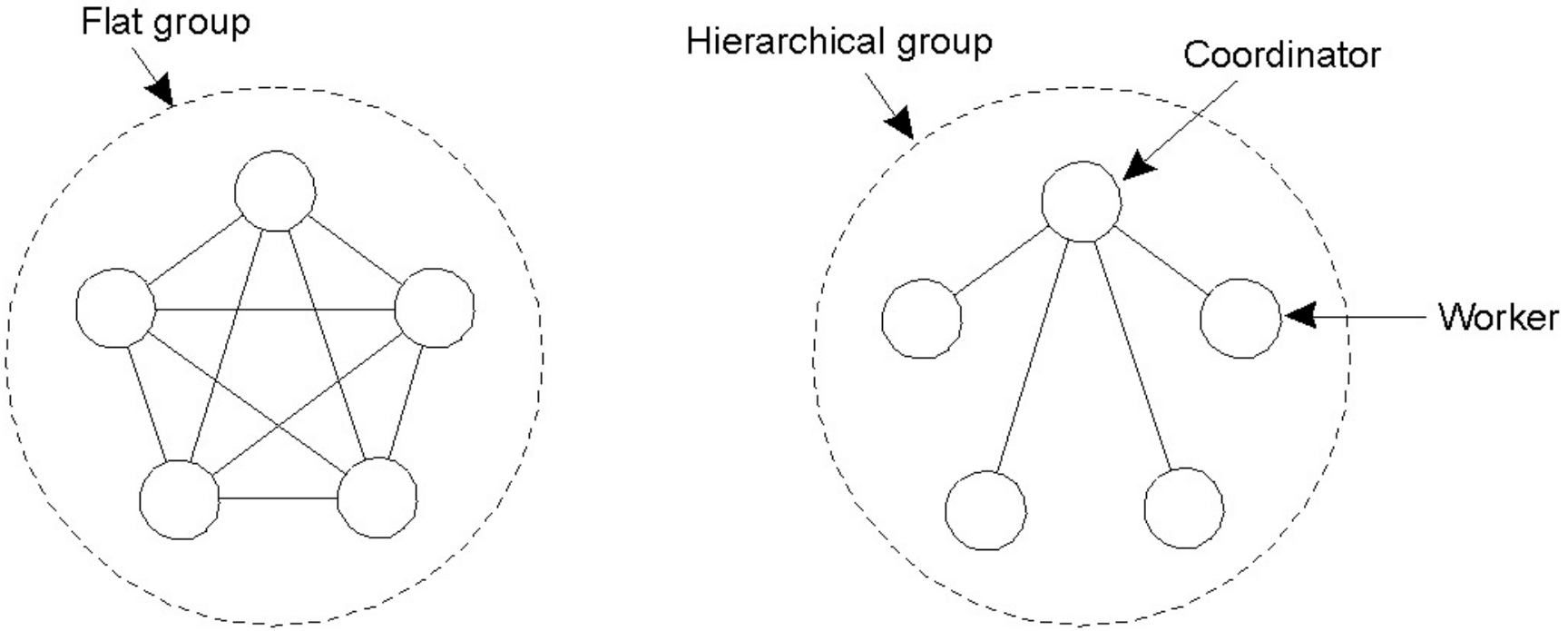
- Classical example: the *state machine* approach
  - make your process a *deterministic state machine*, so that
  - given *n* copies of your process, if you feed them the same inputs:
    1. they will spit out the same outputs
    2. their internal state will be the same



Process resilience

- This poses many questions:
  - Reliable group communications
  - Message ordering
  - Group membership (detection of crashes)
  - Agreement (order of requests, or who is the leader)
- Not all are solvable under all failure modes
- Algorithms (when exist) are useful not only for replication, but also for
  - Distributed mutual exclusion
  - Atomic commitment in distributed DBs
  - etc.

Process groups



- Groups can be organized in two ways
- Centralized decision-making is easier but less reliable
- Collective — might be very difficult (sometimes impossible)

Literature

- G. Coulouris, J. Dollimore, T. Kindberg. Distributed Systems: Concepts and Design (5th edition). Addison-Wesley, 2012
  - Failure modes, [a]synchronous systems, RPC invocation semantics
- Flaviu Cristian. 1991. Understanding fault-tolerant distributed systems. Commun. ACM.
  - Component dependency, failure modes, failure masking and converting, replication
- A. S. Tanenbaum, M. van Steen. Distributed Systems: Principles and Paradigms (2nd edition).
  - Introduction to FT, high-level overview of issues and techniques
- Exactly-Once Delivery May Not Be What You Want
  - http://brooker.co.za/blog/2014/11/15/exactly-once.html