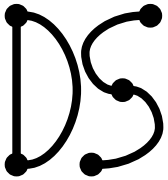


Research Group
Distributed Systems

Distributed Systems

Fault Tolerance I

Olaf Landsiedel

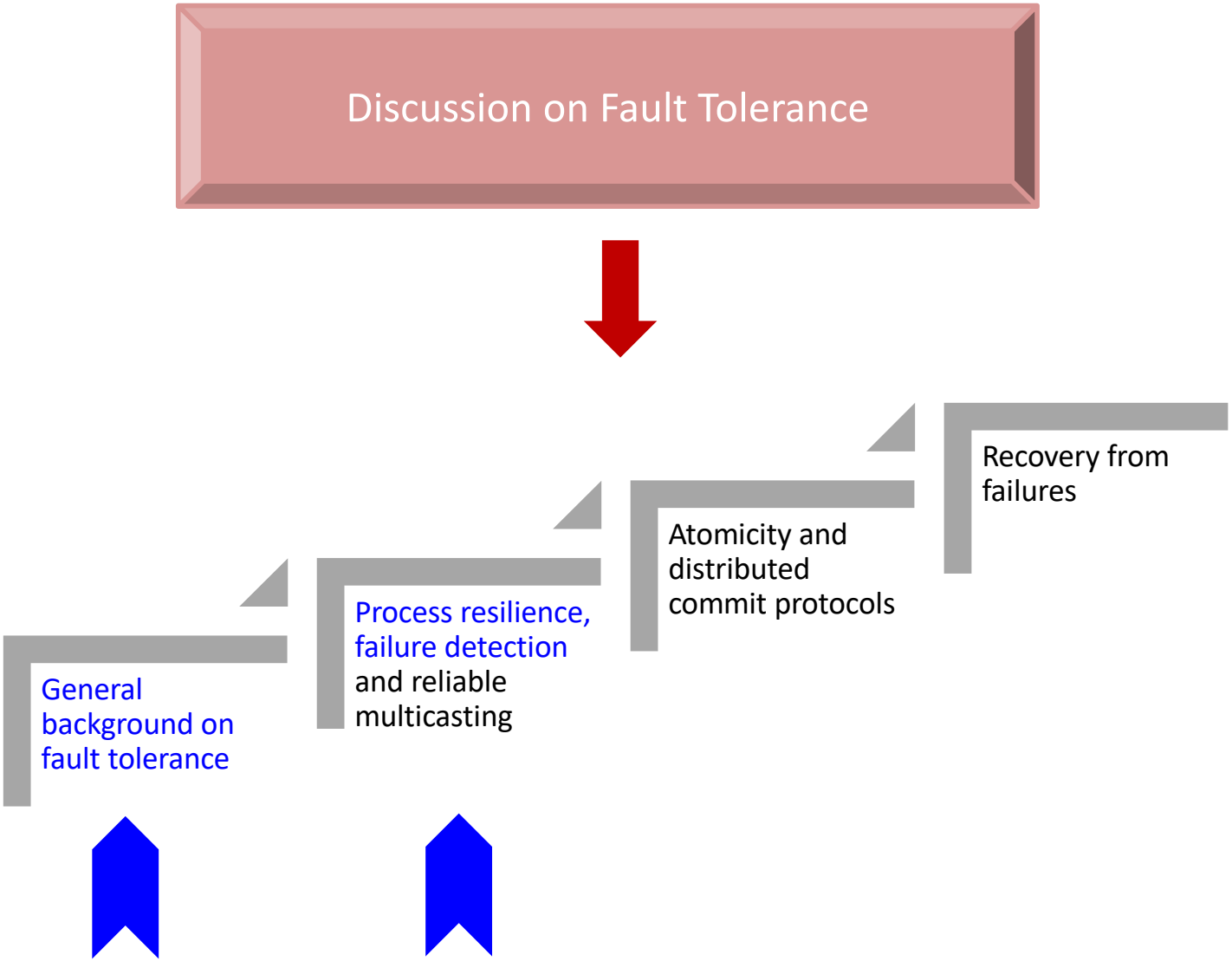
Last Time

- Consistency and Replication III
  - Consistency protocols
  - Case study: Eventual consistency (Bayou)

Until today

- What do you think are the most important things we discussed in this course?

Objectives


Discussion on Fault Tolerance
General background on fault tolerance
Process resilience, failure detection and reliable multicasting
Atomicity and distributed commit protocols
Recovery from failures

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Failures: Example Data Center

- Per year, per cluster (1800 nodes)
  - 1,000 individual machine failures
  - thousands of hard drive failures
  - one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours
  - 20 racks will fail, each time causing 40 to 80 machines to vanish from the network
  - 5 racks will "go wonky," with half their network packets missing in action
  - 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and over

Fault tolerance is key for large-scale distributed systems

n, Google, 2008

Data Center: Hard Disks

- What do you think how many hard disks fail in a data center during a week?
- What is in there? How many nodes (computers)?
  - Modern data center: ~25.000 nodes
- Per node
  - About 4 to 8 cores
  - About 4 HDs
- HD failure rates in data centers?
  - Between 2% and 4% per year
  - 2000 to 4000 HD failures per year
    - 5.5 to

Fault tolerance is key for large-scale distributed systems

Failures, Due to What?

- Failures can happen due to a variety of reasons?
  - Hardware faults
  - Software bugs
  - Operator errors
  - Network errors/outages
  - Power outage

- A system is said to **fail** when it cannot meet its promises

Failures in Distributed Systems

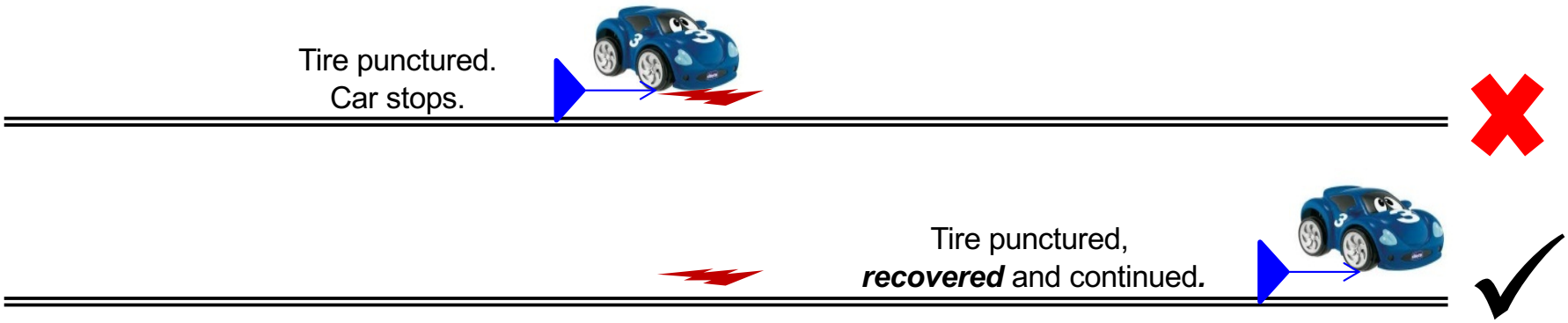
- A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of ***partial failure***

- A partial failure may happen when a component in a distributed system fails

  - This failure may affect the proper operation of other components, while at the same time leaving yet other components unaffected

Goal and Fault-Tolerance

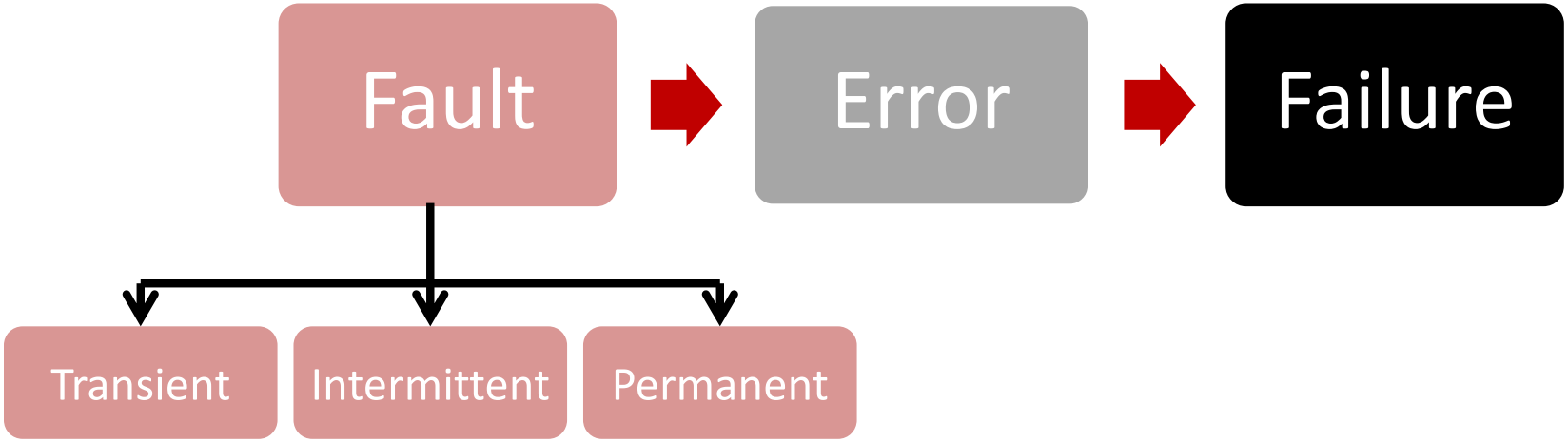
- *An overall goal in distributed systems is to construct the system in such a way that it can automatically recover from partial failures*



- **Fault-tolerance** is the property that enables a system to continue operating properly in the event of failures

- For example, TCP is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communication links which are imperfect or overloaded

Faults, Errors and Failures



A system is said to be *fault tolerant* if it can provide its services even in the presence of *faults*

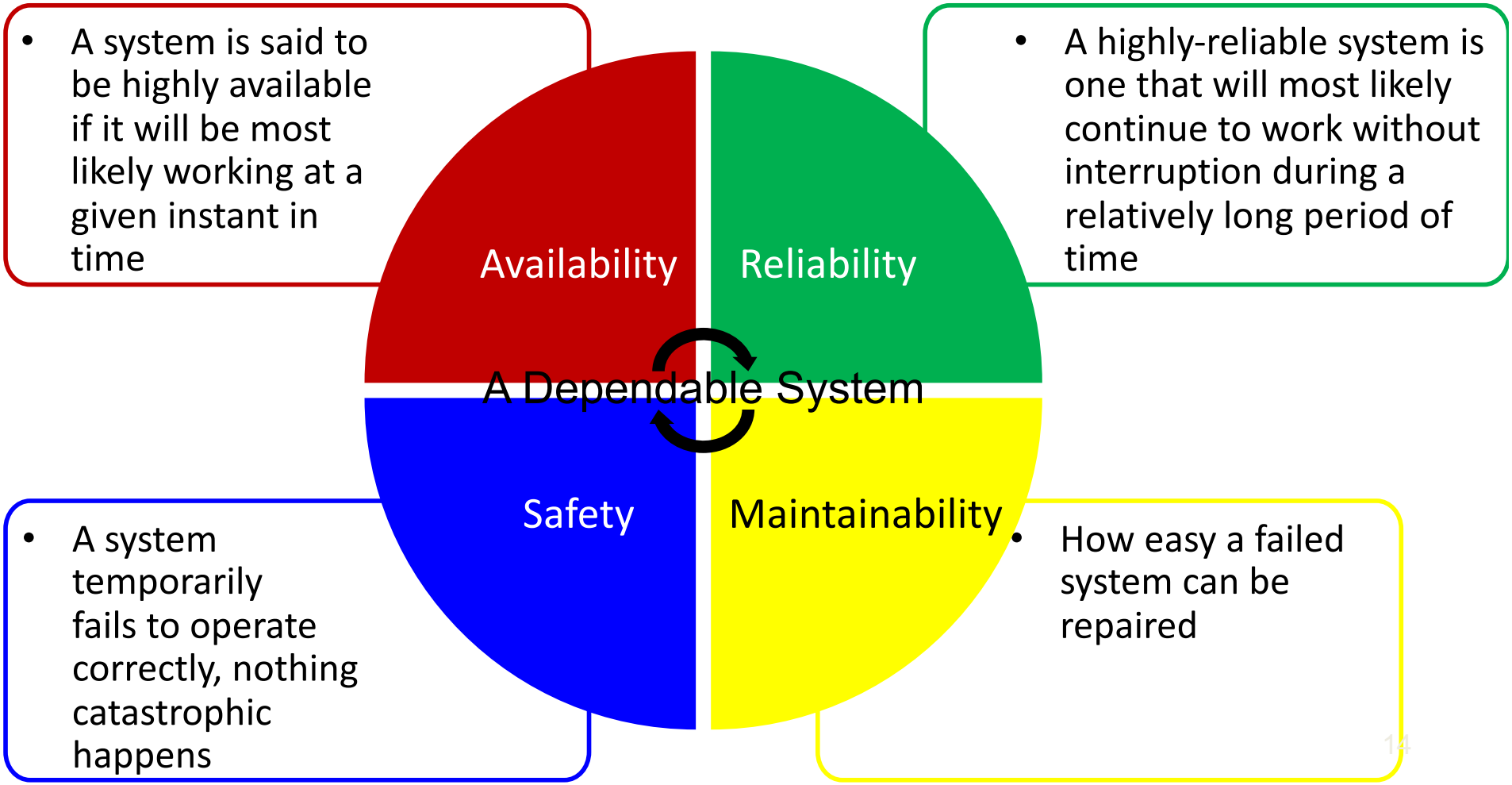
Fault Tolerance Requirements

- A robust fault tolerant system requires?

  1. No single point of failure
  2. Fault isolation/containment to the failing component
  3. Availability of reversion modes

Dependable Systems

- Being fault tolerant is strongly related to what is called a *dependable system*



A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Failure Models

Type of Failure	Description
• Crash Failure	• A server halts, but was working correctly until it stopped

• Omission Failure
  • Receive Omission
  • Send Omission
	• A server fails to respond to incoming requests
	  • A server fails to receive incoming messages
	  • A server fails to send messages

• Timing Failure	• A server's response lies outside the specified time interval

• Response Failure
  • Value Failure
  • State Transition Failure
	• A server's response is incorrect
	  • The value of the response is wrong
	  • The server deviates from the correct flow of control

• Byzantine Failure	• A server may produce arbitrary responses at arbitrary times

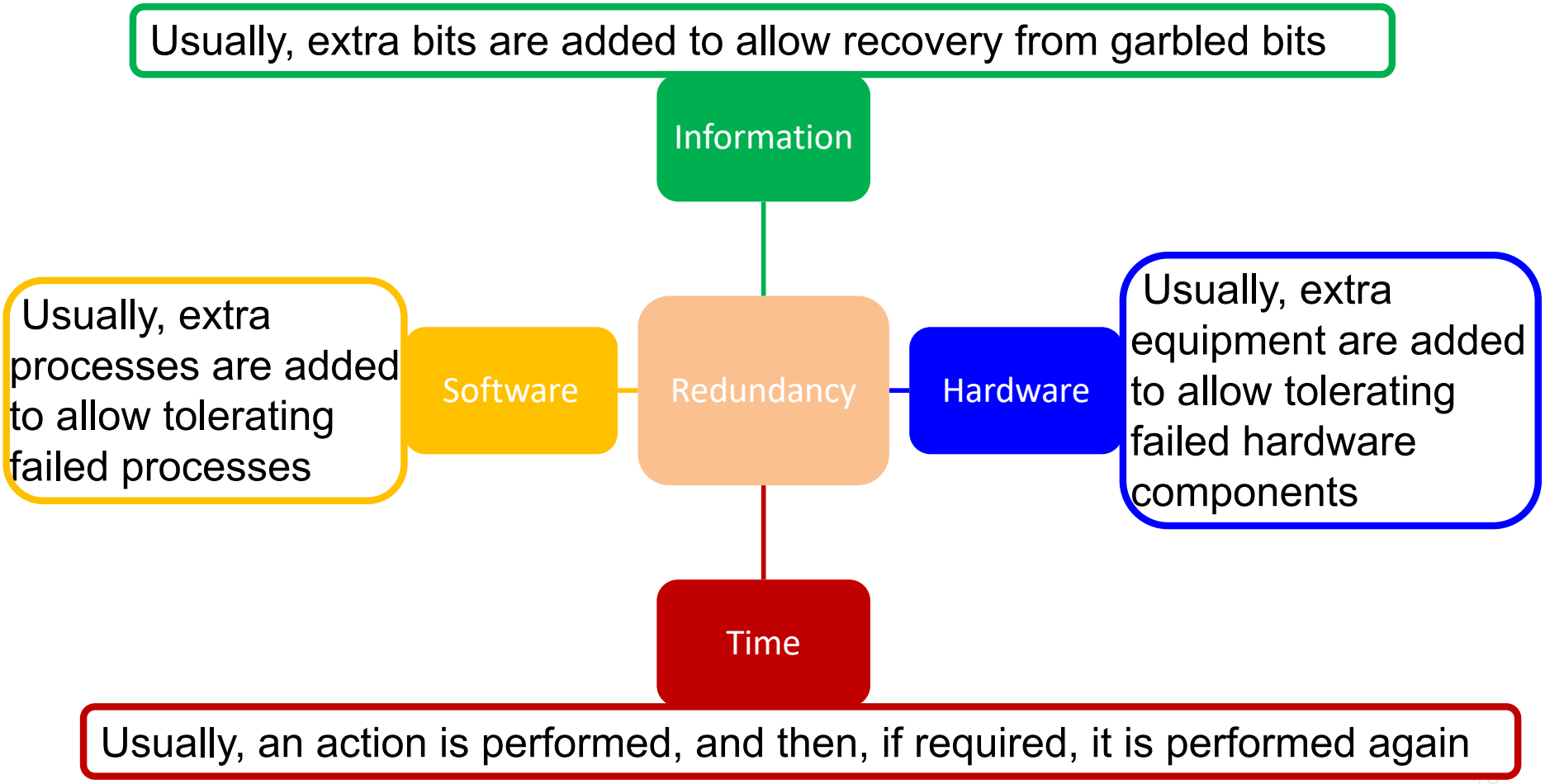
A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Faults Masking by Redundancy

- The key technique for masking faults is to use *redundancy*

Examples?



Redundancy Examples

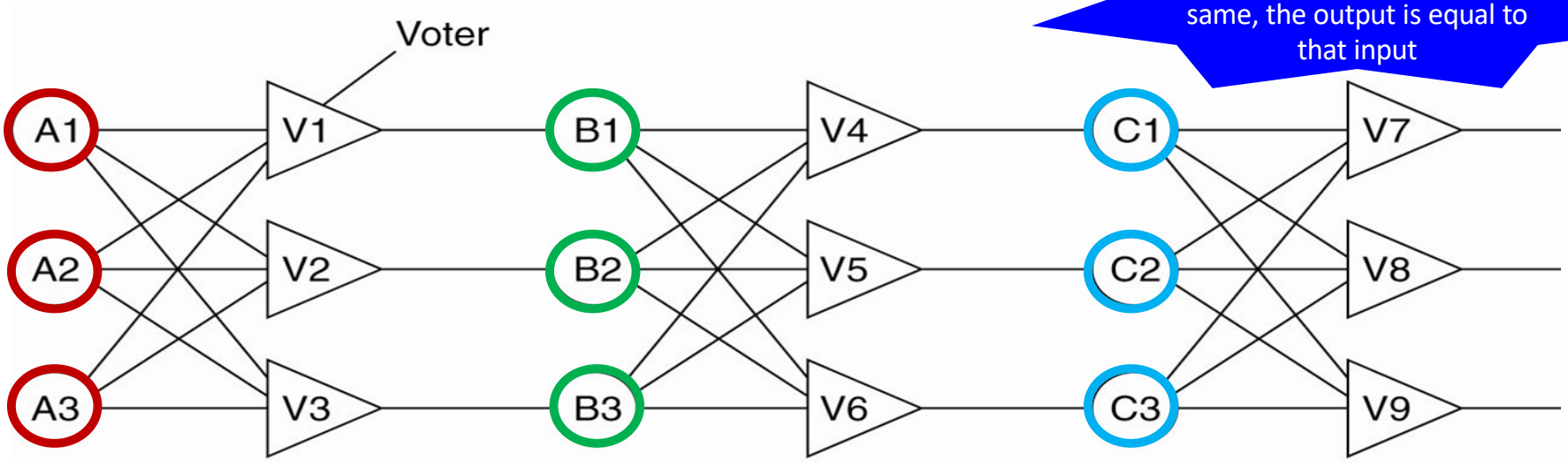
- Information
  - Link layer of wireless communication: coding schemes add redundancy to make system immune to interference
  - same for CDs and DVDs
- Hardware
  - multiple power supplies, HDs, etc.
- Software
  - Critical Systems: Nuclear Power stations, airplanes etc:
  - run the control software concurrently on multiple systems
- Time
  - this is rare, but of course I can also run the same software multiple times

Triple Modular Redundancy

If one is faulty, the final result will be incorrect

A circuit with signals passing through devices A, B, and C, in sequence

If 2 or 3 of the inputs are the same, the output is equal to that input



Each device is replicated 3 times and after each stage is a triplicated voter

Questions

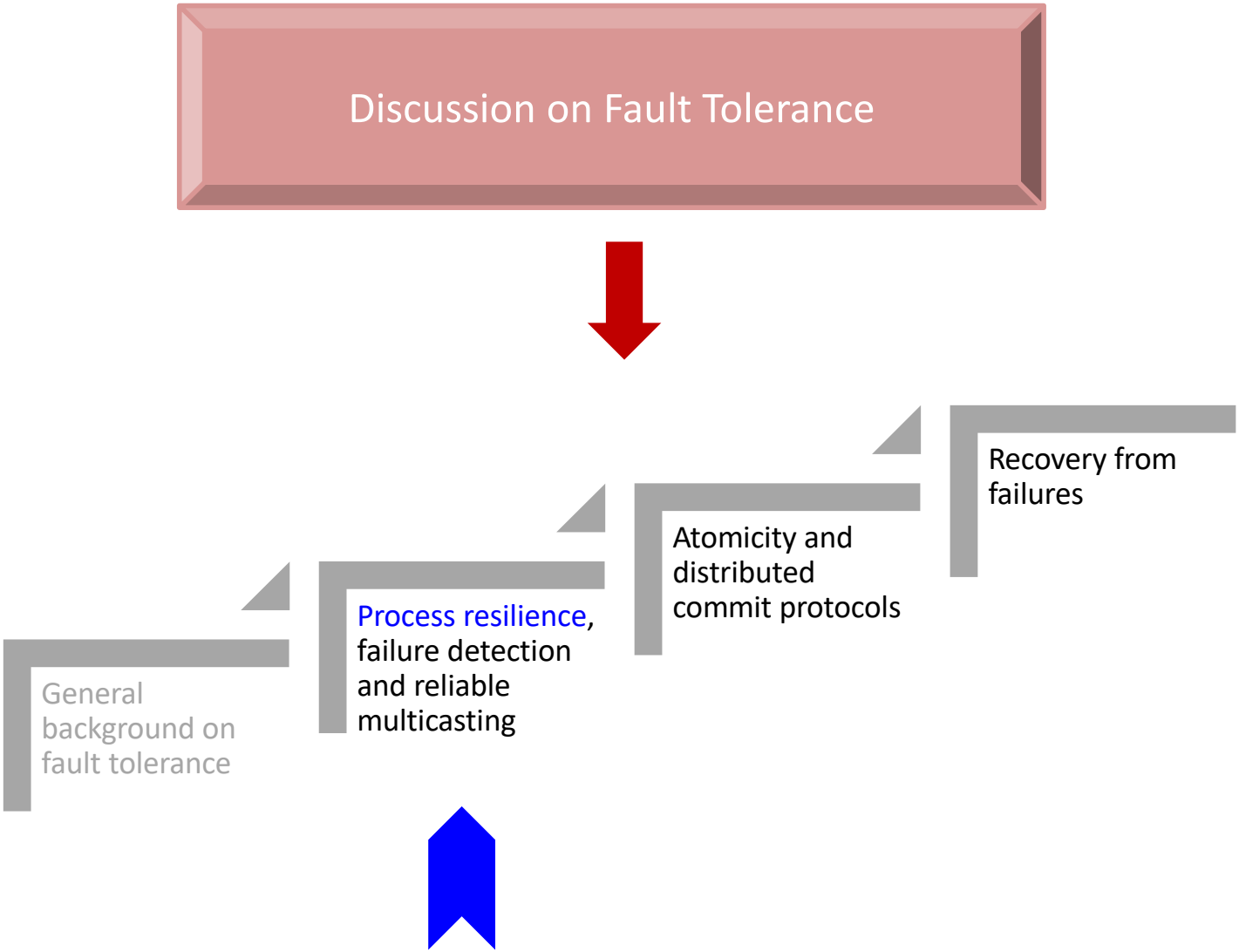
- How to build systems that can deal with failures?
  - Avoid single point of failure
  - Fault isolation/containment to the failing component
  - Availability of reversion modes

- http://olafland.polldaddy.com/s/data-center-failure
  - Which fault is the hardest to debug (Transient, Intermittent, Permanent)?
  - Guess, what is the most common reason for data center failure (e.g., most services unavailable)?
  - Guess, what is the second most common reason for data center failure (e.g., most services unavailable)?

Answers

- Which fault is the hardest to debug (Transient, Intermittent, Permanent)?
  - Intermittent is the hardest to debug: it very challenging to reproduce
  - (transient ones are often not debugged: happen only once)
    - Why:
    - computers are redundant with good failure protocols for software
    - computers are failing constantly (see prev. slides) so software and hardware failure over protocols are getting constantly tested
- Guess, what is the most common reason for data center failure (e.g., most services unavailable)?
  - Power outage
- Guess, what is the second most common reason for data center failure (e.g., most services unavailable)?
  - Human error
- Regarding the data center failures
  - You find many different sources (and some with other results and/or other application settings, i.e., not data centers)
  - Here are two sources I used
  - http://www.emersonnetworkpower.com/documents/en-us/brands/liebert/documents/white%20papers/data-center-optimization_24655-r09-10.pdf
  - http://www.netmagicsolutions.com/uploads/pdf/resources/whitepapers/WP_Datacenter-Outages.pdf

Objectives


Discussion on Fault Tolerance
General background on fault tolerance
Process resilience, failure detection and reliable multicasting
Atomicity and distributed commit protocols
Recovery from failures

PROCESS RESILIENCE AND FAILURE DETECTION

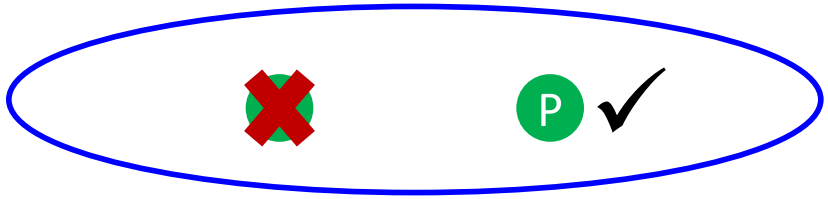
Process Resilience and Failure Detection

- Now that the basic issues of fault tolerance have been discussed, let us concentrate **on how fault tolerance can actually be achieved in distributed systems**

- The topics we will discuss:
  - How can we provide protection against process failures?
    - Process groups
    - Reaching an agreement within a process group
  - How to detect failures?

Process Resilience

- The key approach to tolerating a faulty process is to organize several identical processes into a *group*



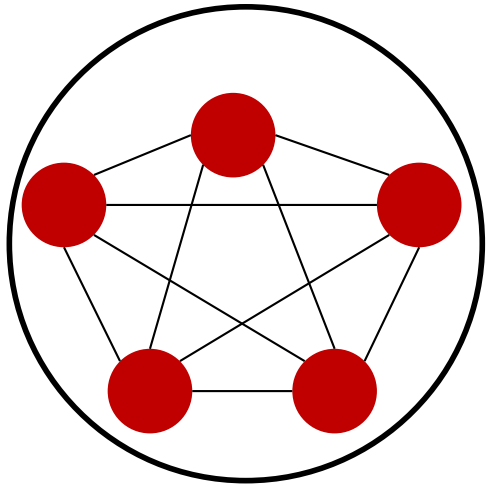
- If one process in a group fails, hopefully some other process can take over

- Caveats:

  - A process can join a group or leave one during system operation
  - A process can be a member of several groups at the same time

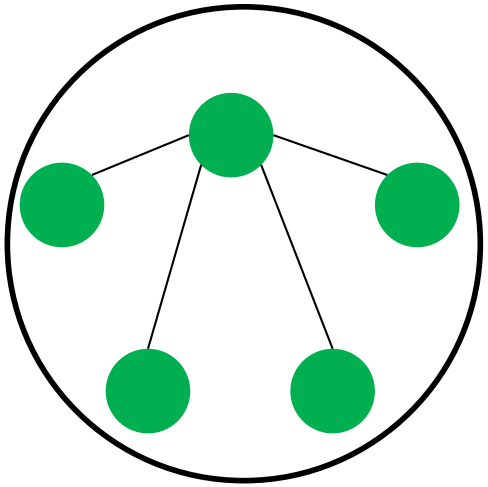
Flat Versus Hierarchical Groups

- An important distinction between different groups has to do with their internal structure



Flat Group:
e.g. quorum based

(+) Symmetrical
(+) No single point of failure
(-) Decision making is complicated



Hierarchical Group:
e.g. active replication

(+) Decision making is simple
(-) Asymmetrical
(-) Single point of failure

K-Fault-Tolerant Systems

- A system is said to be *k-fault-tolerant* if it can survive faults in *k* components and still meet its specifications

- How can we achieve a *k-fault-tolerant* system?

  - This would require an *agreement protocol* applied to a process group

Agreement in Faulty Systems (1)

- Example: A process group typically requires reaching an *agreement* in:
  - Electing a coordinator
  - Deciding whether or not to commit a transaction
  - Dividing tasks among workers
  - Synchronization

- When the communication and processes:
  - are perfect, reaching an agreement is often straightforward
  - are not perfect, there are problems in reaching an agreement

Agreement in Faulty Systems (2)

- **Goal:** have all non-faulty processes reach consensus on some issue, and establish that consensus within a finite number of steps

- Different assumptions about the underlying system require different solutions:

  - Synchronous versus asynchronous systems
  - Communication delay is bounded or not
  - Message delivery is ordered or not
  - Message transmission is done through unicasting or multicasting

Agreement in Faulty Systems (3)

- Reaching a distributed agreement is only possible in the following circumstances:

		Message Ordering					
		Unordered		Ordered			
Process Behavior	Synchronous	✓	✓	✓	✓	Bounded	Communication Delay
				✓	✓	Unbounded	
	Asynchronous				✓	Bounded	
					✓	Unbounded	
		Unicast	Multicast	Unicast	Multicast		
		Message Transmission					

Agreement in Faulty Systems (4)

- In practice most distributed systems assume that:
  - Processes behave asynchronously
  - Message transmission is unicast
  - Communication delays are unbounded
- Usage of ordered (reliable) message delivery is typically required
- The agreement problem has been originally studied by Lamport and referred to as the *Byzantine Agreement Problem*

Questions

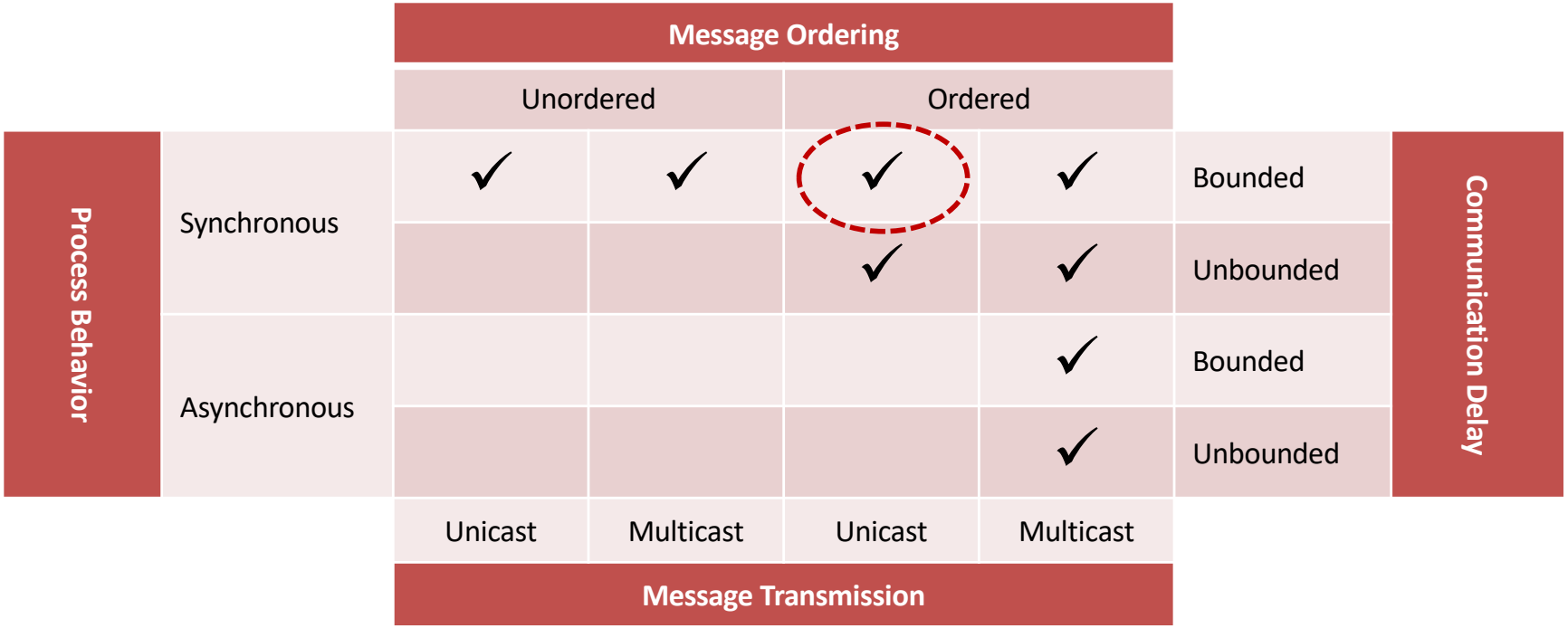
- Challenge with unbounded communication delay?
  - How to distinguish between long network delay and failure
- Why do we consider computer to be asynchronous and not synchronous?
  - Synchronous system require strong time synchronization
- Why do we consider networks to be unicast (and not multicast)?
  - Multicast is challenging (group membership etc.)
- Which protocol gives us ordered messages?
  - TCP (sequence numbers)

Byzantine Agreement Problem (1)

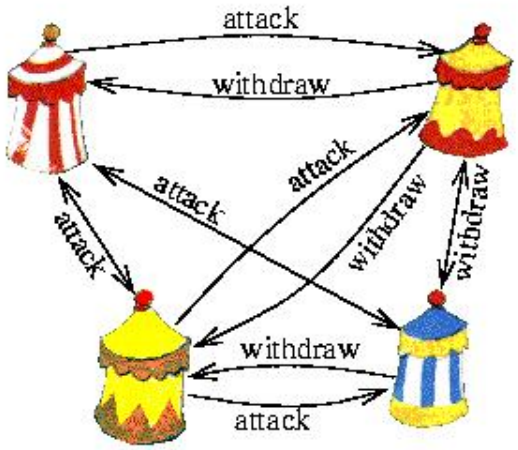
- We assume
  - Processes are synchronous
  - Messages are unicast while preserving ordering
  - Communication delay is bounded
  - There are **N** processes, where each process **i** will provide a value $\mathbf{v_i}$ to the others
  - There are at most **k** faulty processes

- Example for Byzantine Agreement Problem
  - Byzantine Generals Problem
  - Also known as: Three Generals Problem

Byzantine Agreement Problem (2)



Byzantine Agreement Problem (3)

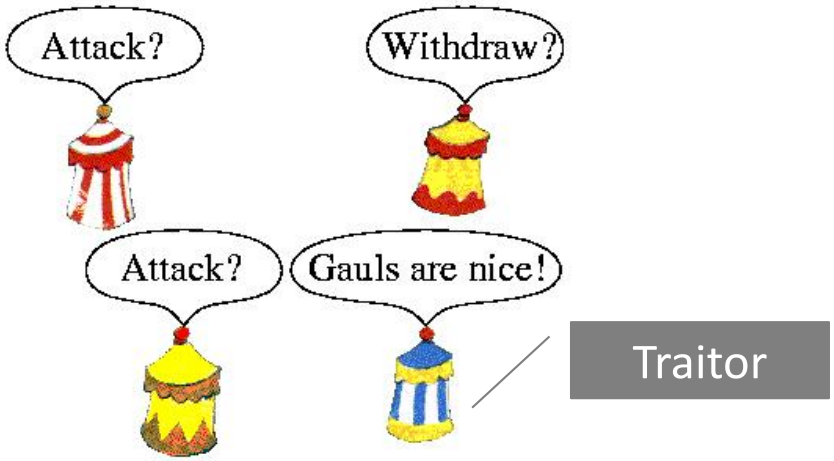


- Generals (**N** of them) surround a city
  - They communicate by courier
- Each has an opinion: "attack" or "wait (withdraw)"
  - In fact, an attack would succeed: the city will fall.
  - Waiting will succeed too: the city will surrender.
  - But if some attack and some wait, disaster ensues
  - -> Must reach consensus
- Some Generals (**f** of them) are traitors… it doesn't matter if they attack or wait, but we must prevent them from disrupting the battle
  - Traitor can send different messages to each
  - Traitor can wait to see what others say
  - Traitor can't forge messages from other Generals
  - Traitors = "Faulty Processes"

The pictures are taken from: R. Goscinny and A. Uderzo, Asterix and Latraviata.

Byzantine Agreement Problem (4)

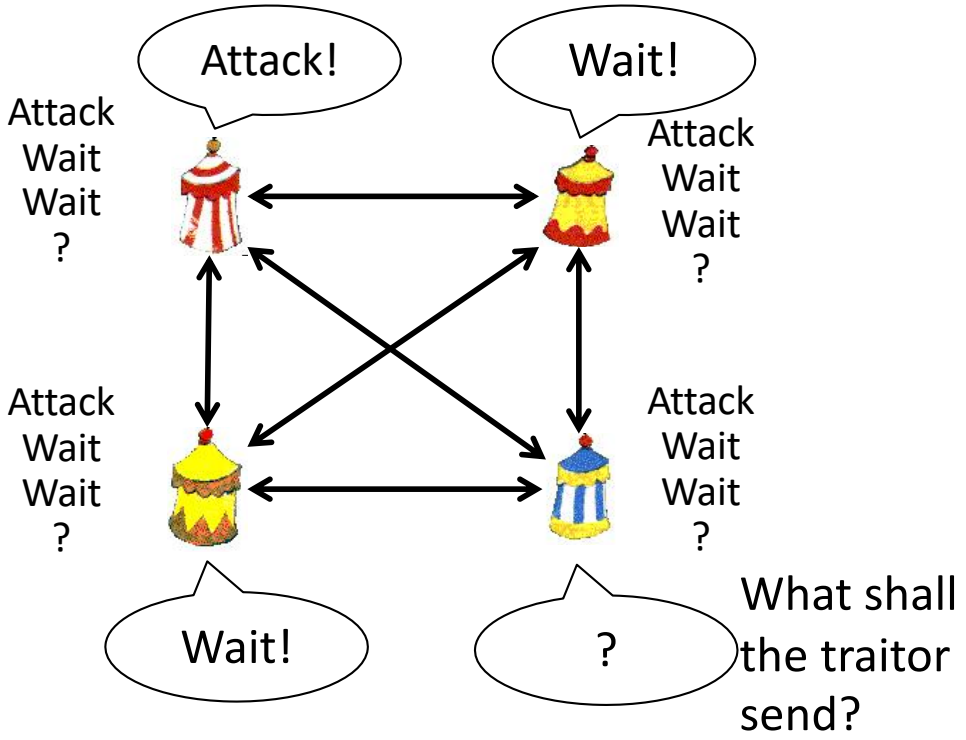
Some of them may be traitors who will try to confuse the others

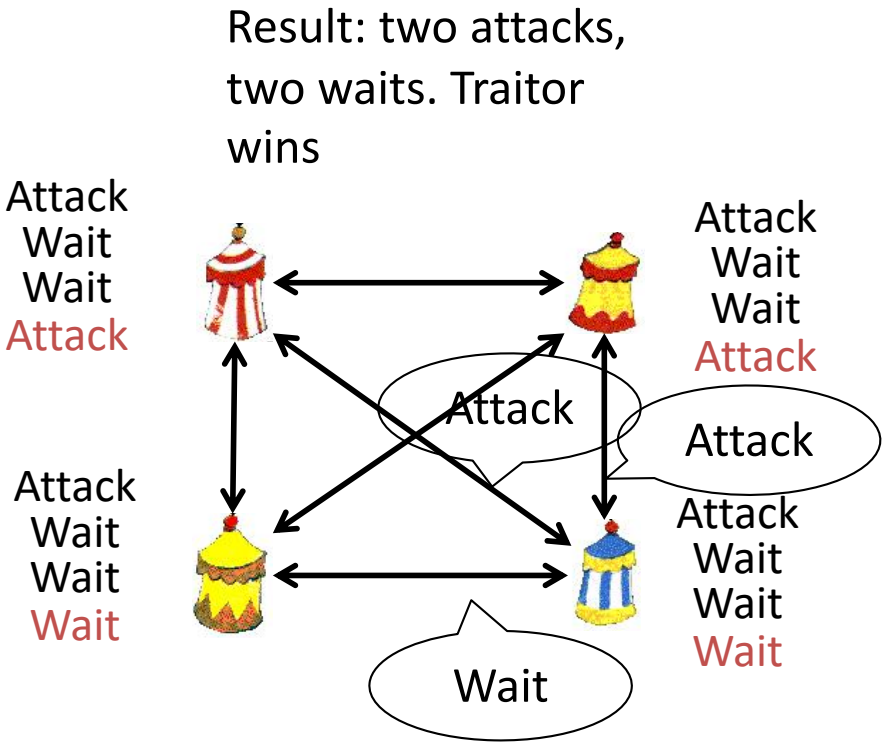


- What “protocol” or “algorithm” shall the generals use
  - To ensure that they reach consensus
  - -> Generals win, traitor(s) loose

Byzantine Agreement Problem (5)

- Example: 4 generals, 1 is a traitor (blue tent)
  - On tie: we attack





- What "protocol" or "algorithm" shall the generals use
  - To ensure that they reach consensus
  - -> Generals win, traitor(s) loose

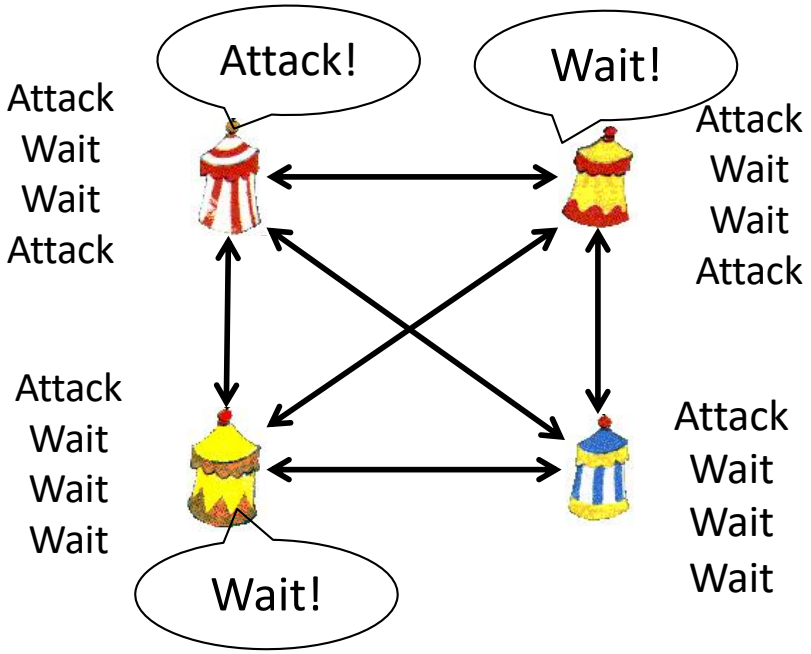
Byzantine Agreement Problem (6)

- Basic Idea: 2 step protocol
  1. Generals send votes to each other
  2. Exchange what each general got from the others
     - After all votes have been received

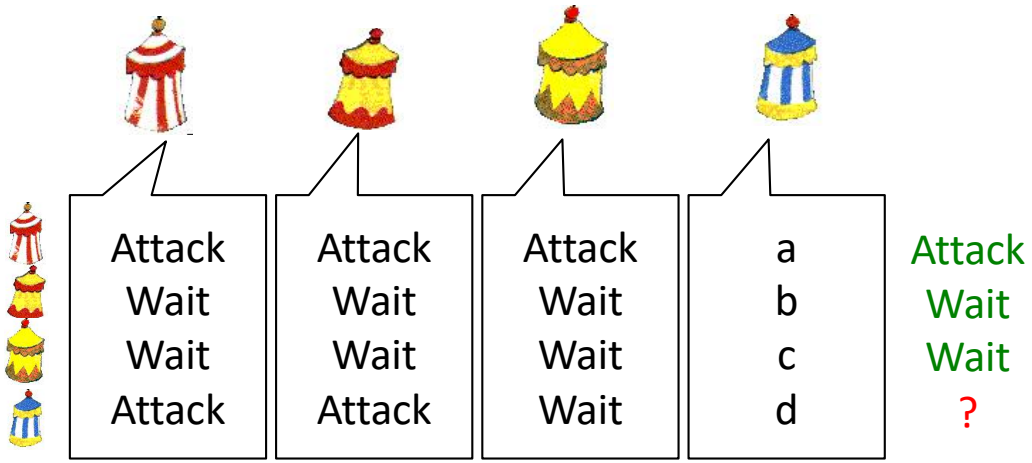
- Idea
  - This should help to “filter out” the “faulty” processes

Byzantine Agreement Problem (7)

- Step 2: exchange rec. commands

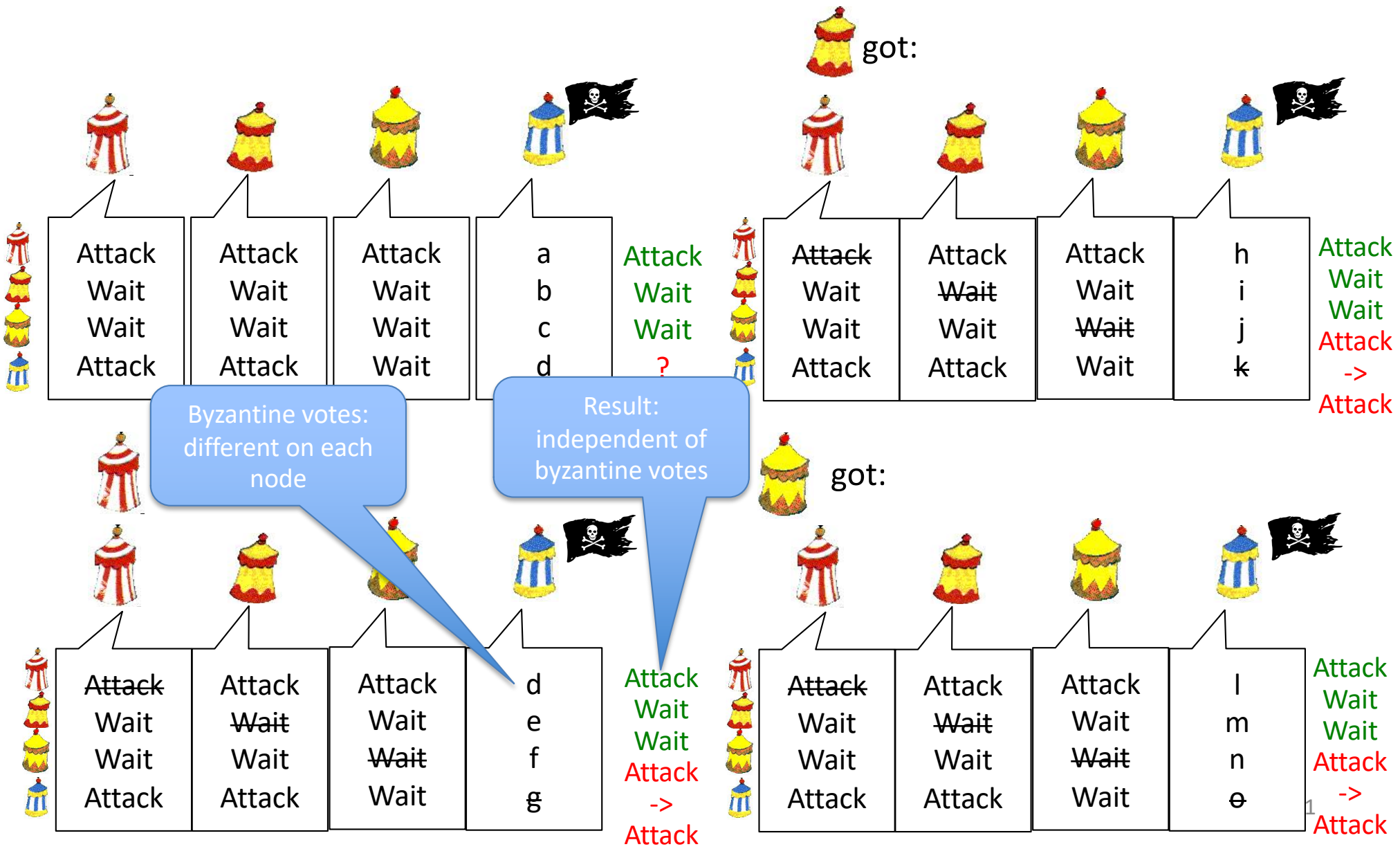


After exchange: what each node received



- Let's discuss in more detail….

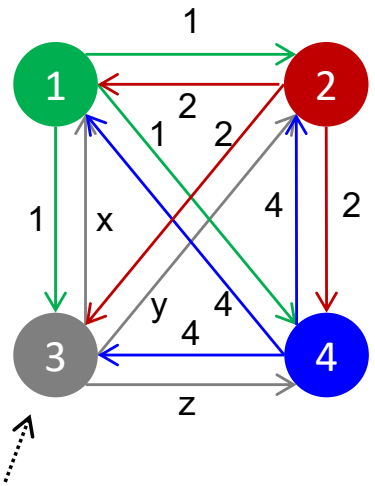
Byzantine Agreement Problem (8)



Byzantine Agreement Problem (9)

- **Case I: N = 4 and k = 1**

Step1: Each process sends its value to the others



Faulty process

Step2: Each process collects values received in a vector

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

Step3: Every process passes its vector to every other process

1 Got

(1, 2, x, 4)
(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

2 Got

(1, 2, x, 4)
(1, 2, y, 4)
(e, f, g, h)
(1, 2, z, 4)

4 Got

(1, 2, x, 4)
(1, 2, y, 4)
(i, j, k, l)
(1, 2, z, 4)

Byzantine Agreement Problem (10)

Step 4:

- Each process examines the *i*th column of each of the received vectors, crosses out diagonal (as in previous example)
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

1 Got

(~~1~~, 2, x, 4)
(1, ~~2~~, y, 4)
(a, b, ~~c~~, d)
(1, 2, z, 4)

Result Vector:
(1, 2, UNKNOWN, 4)

2 Got

(~~1~~, 2, x, 4)
(1, ~~2~~, y, 4)
(e, f, ~~g~~, h)
(1, 2, z, 4)

Result Vector:
(1, 2, UNKNOWN, 4)

4 Got

(~~1~~, 2, x, 4)
(1, ~~2~~, y, 4)
(i, j, ~~k~~, l)
(1, 2, z, 4)

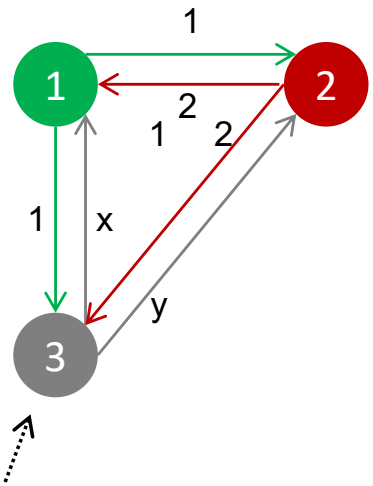
Result Vector:
(1, 2, UNKNOWN, 4)

The algorithm reaches an agreement

Byzantine Agreement Problem (11)

- **Case II: N = 3 and k = 1**

Step1: Each process sends its value to the others



Faulty process

Step2: Each process collects values received in a vector

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

Step3: Every process passes its vector to every other process

1 Got

(1, 2, x)
(1, 2, y)
(a, b, c)

2 Got

(1, 2, x)
(1, 2, y)
(d, e, f)

Byzantine Agreement Problem (12)

Step 4:

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

1 Got

(~~1~~, 2, x)
(1, ~~2~~, y)
(a, b, ~~c~~)

The algorithm has failed to produce an agreement

2 Got

(~~1~~, 2, x)
(1, ~~2~~, y)
(d, e, ~~f~~)

Result Vector:
(UNKOWN, UNKNOWN, UNKNOWN)

Result Vector:
(UNKOWN, UNKNOWN, UNKNOWN)

Summary

Concluding Remarks on the Byzantine Agreement Problem

- In their paper, *Lamport et al.* (1982) proved that in a system with ***k*** faulty processes, an agreement can be achieved only if ***2k+1*** correctly functioning processes are present, for a total of ***3k+1***.
  - i.e., An agreement is possible only if more than two-thirds of the processes are working properly.
  - Moreover, for the algorithms shown, we need k+1 rounds
    - In our example k was 1. Thus, 2 phase (or rounds) are required
- *Fisher et al.* (1985) proved that in a distributed system in which ordering of messages **cannot** be guaranteed to be delivered within a known, finite time, no agreement is possible even if only one process is faulty.

Questions

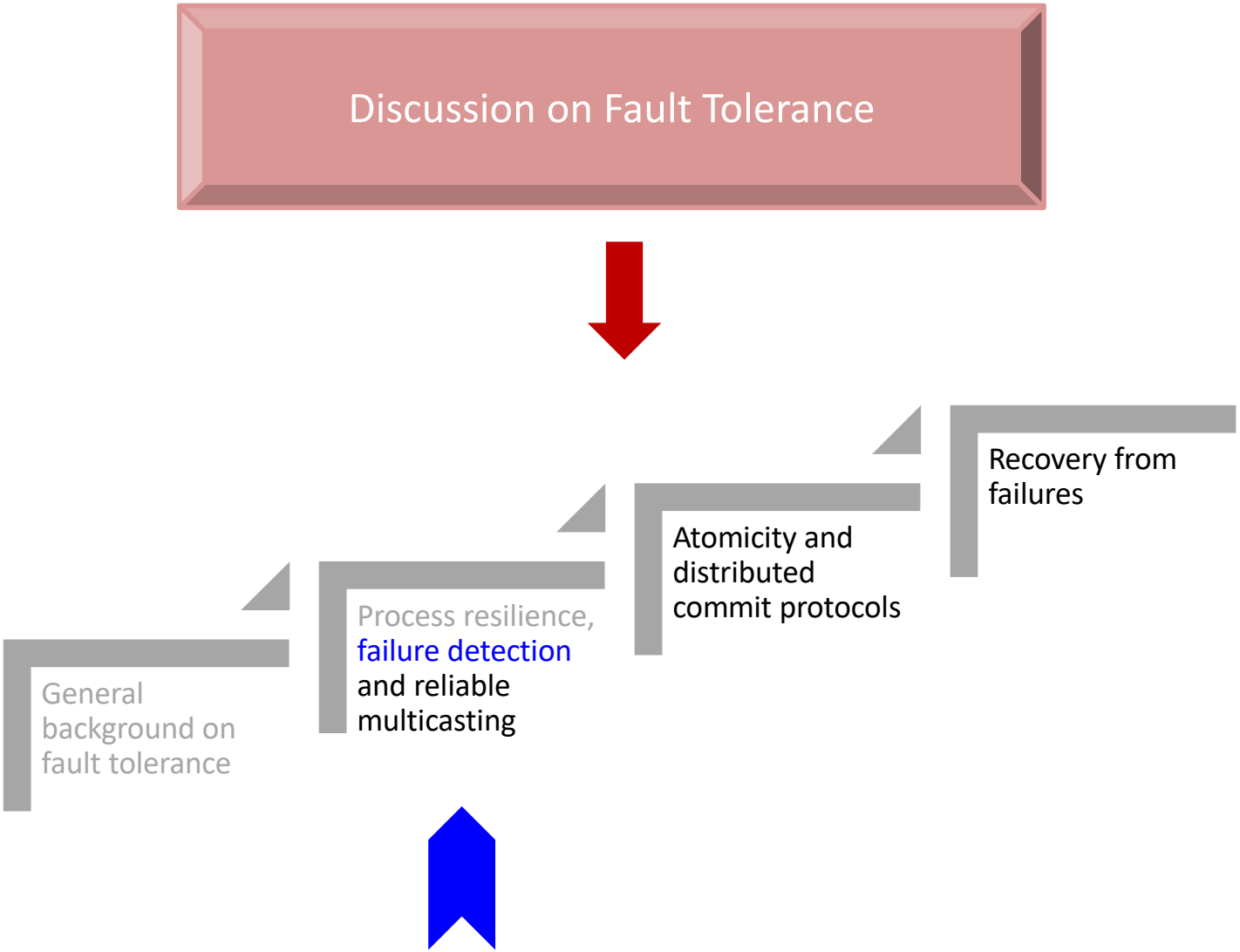
- Summarize the Byzantine Agreement Problem

- http://olafland.polldaddy.com/s/byzantine-agreement-problem
  - The algorithm is easy to implement
  - The algorithm has a single point of failure
  - The algorithm will work fine for any number of “traitors”
  - With the algorithm we can detect who is the traitor
  - The algorithm is very scalable
  - What is the message complexity (k=1)?

Answers

- The algorithm is easy to implement: no
- The algorithm has a single point of failure: no
- The algorithm will work fine for any number of “traitors”: no
- With the algorithm we can detect who is the traitor: no
- What is the message complexity? O(n*n)

Objectives


Discussion on Fault Tolerance
General background on fault tolerance
Process resilience, failure detection and reliable multicasting
Atomicity and distributed commit protocols
Recovery from failures

Process Failure Detection

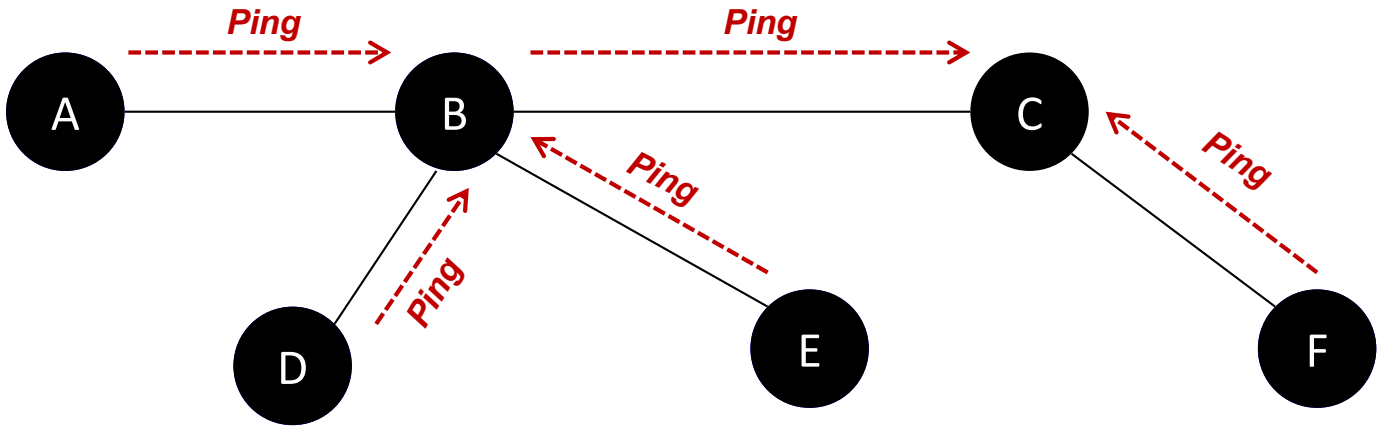
- Before we properly mask failures, we generally need to detect them
- For a group of processes, non-faulty members should be able to decide who is still a member and who is not
- Two policies:
  - Processes actively send "are you alive?" messages to each other (i.e., *pinging each other*)
  - Processes passively wait until messages come in from different processes

Timeout Mechanism

- In failure detection a ***timeout mechanism*** is usually involved
  - Specify a timer, after a period of time, trigger a timeout
    - Best known example?
      - TCP
  - Problem?
    - However, due to unreliable networks, simply stating that a process has failed because it does not return an answer to a ping message may be wrong

Example: FUSE

- In FUSE, processes can be joined in a group that spans a WAN
  - If one fails, others shall switch to fail state, too
- The group members create a spanning tree that is used for monitoring member failures
- An active (pinging) policy is used where a single node failure is rapidly promoted to a group failure notification



Failed Member Alive Member

Failure Considerations

- There are various issues that need to be taken into account when designing a failure detection subsystem:

  1. Failure detection can be done as a side-effect of regularly exchanging information with neighbors (e.g., ***gossip-based information dissemination***)

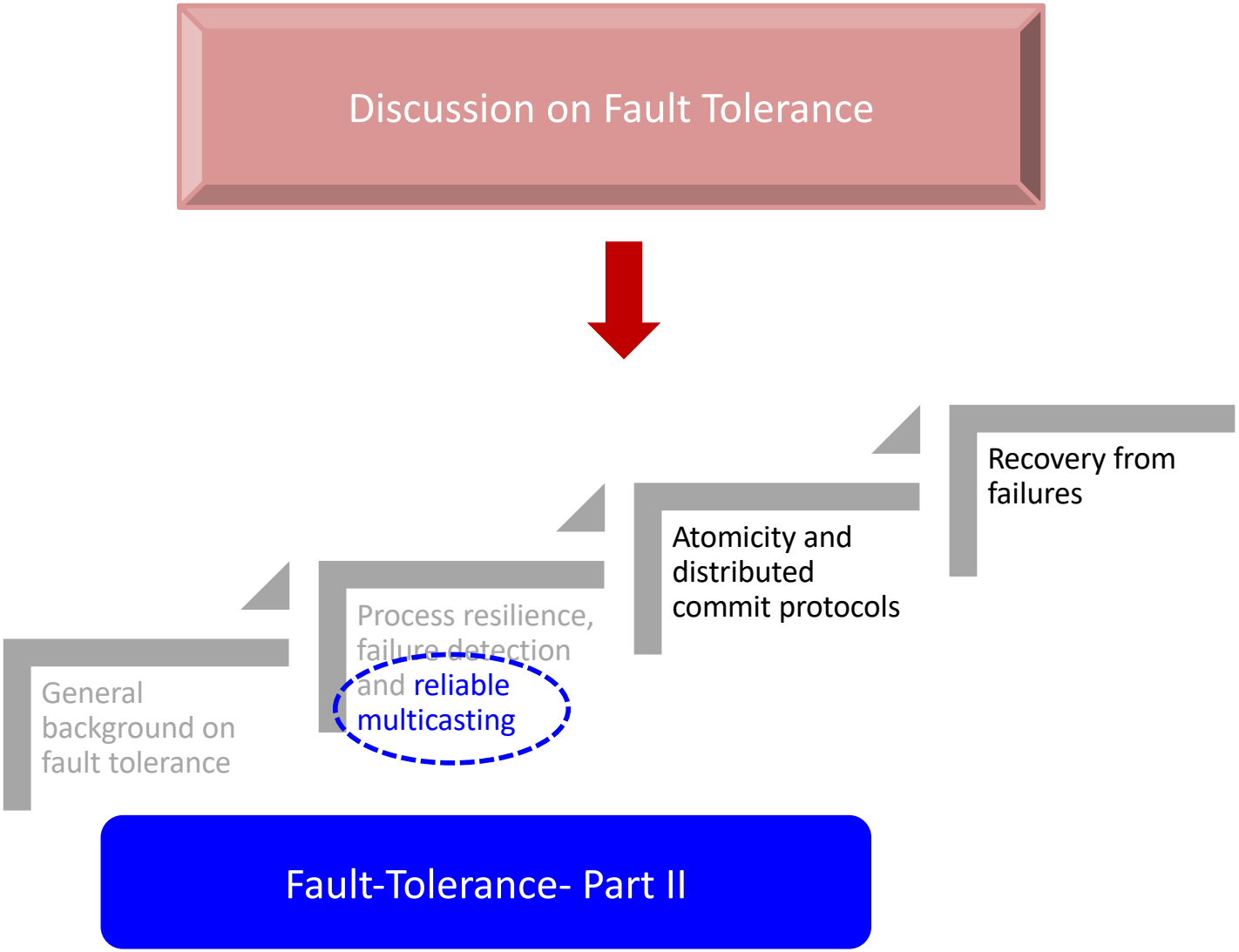
  2. A failure detection subsystem should ideally be able to distinguish network failures from node failures

  3. When a member failure is detected, how should other non-faulty processes be informed

Summary

- Always design for failures
  - See your labs, see data centers
- Often the reason for failure is different than expected
  - See data centers
- Algorithms that can deal well with failures
  - Are commonly quite complex

Next Class


Discussion on Fault Tolerance
General background on fault tolerance
Process resilience, failure detection and reliable multicasting
Atomicity and distributed commit protocols
Recovery from failures
Fault-Tolerance- Part II

Questions?

In part, inspired from / based on slides from

- Mohammad Hammoud
- Muyuan Wang