Unit-8
Fault Tolerance

Compiled by Prashant Gautam

Objectives

Discussion on Fault Tolerance

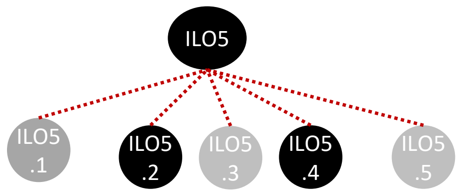
- General background on fault tolerance
- Process resilience, failure detection and reliable multicasting
- Atomicity and distributed commit protocols
- Recovery from failures

Intended Learning Outcomes

○ ***Considered:*** a reasonably critical and comprehensive perspective.

◐ ***Thoughtful:*** Fluent, flexible and efficient perspective.

● ***Masterful:*** a powerful and illuminating perspective.



ILO5	Explain how a distributed system can be made fault tolerant
ILO5.1	Describe dependable systems, different types of failures, and failure masking by redundancy
ILO5.2	Explain how process resilience can be achieved in a distributed system
ILO5.3	Describe the five different classes of failures that can occur in RPC systems
ILO5.4	Explain different schemes of reliable multicasting, scalability in reliable multicasting, the atomic multicast problem, the distributed commit problem, and virtual synchrony
ILO5.5	Describe when and how the state of a distributed system can be recorded and recovered

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Failures, Due to What?

- Failures can happen due to a variety of reasons:
  - Hardware faults
  - Software bugs
  - Operator errors
  - Network errors/outages
- A system is said to *fail* when it cannot meet its promises

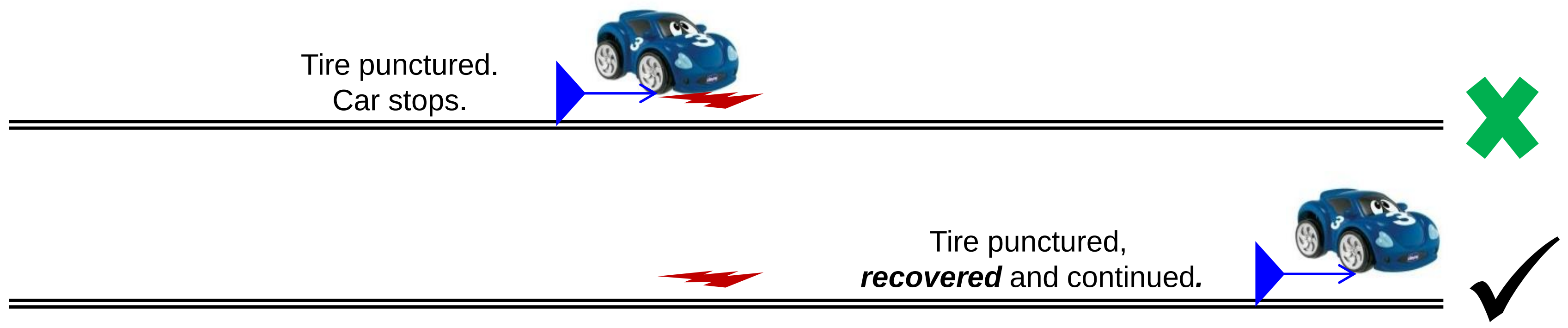
Failures in Distributed Systems

- 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

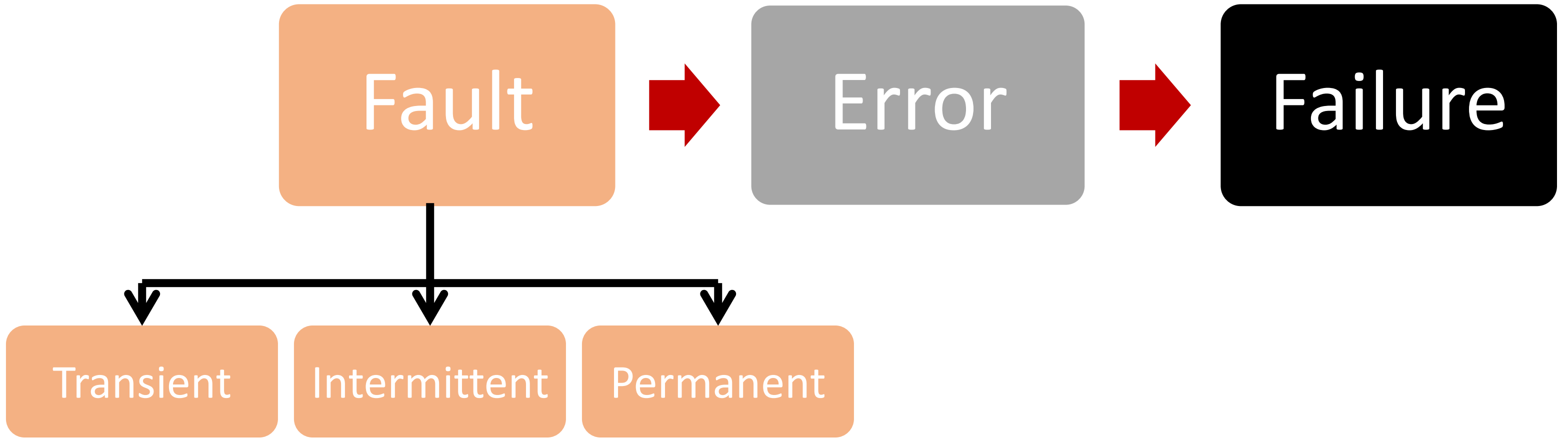
- *An overall goal in distributed systems is to construct the system in such a way that it can automatically recover from partial failures*



- **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:

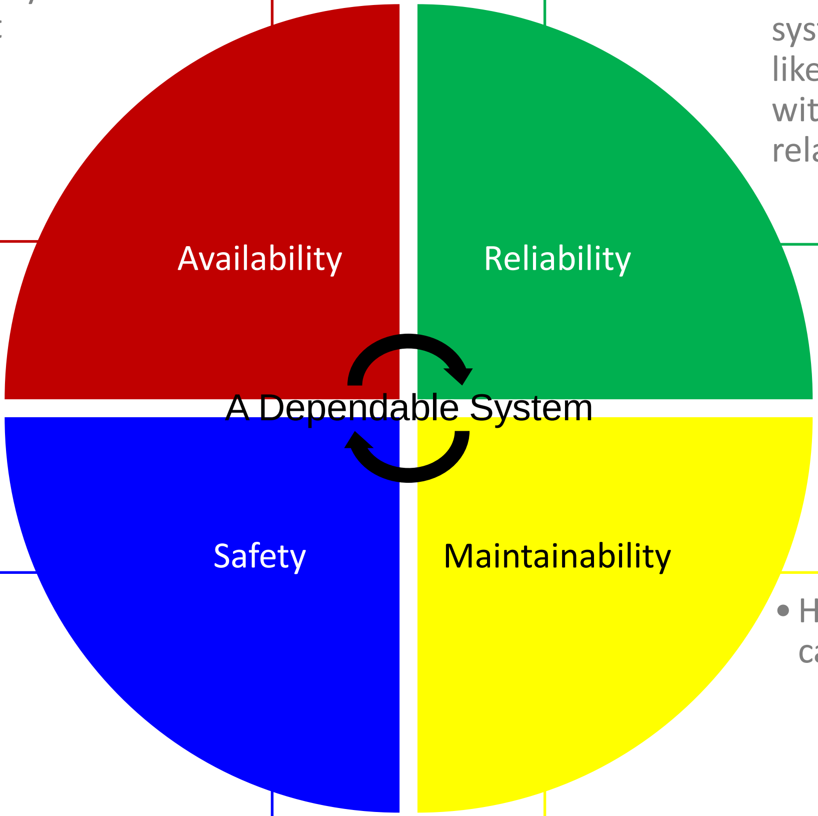
    1. No single point of failure
    2. Fault isolation/containment to the failing component
    3. Availability of reversion modes

Dependable Systems

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

- A system is said to be highly available if it will be most likely working at a given instant in time

- A highly-reliable system is one that will most likely continue to work without interruption during a relatively long period of time



- A system temporarily fails to operate correctly, nothing catastrophic happens

- How easy a failed system can be repaired

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Failure Models

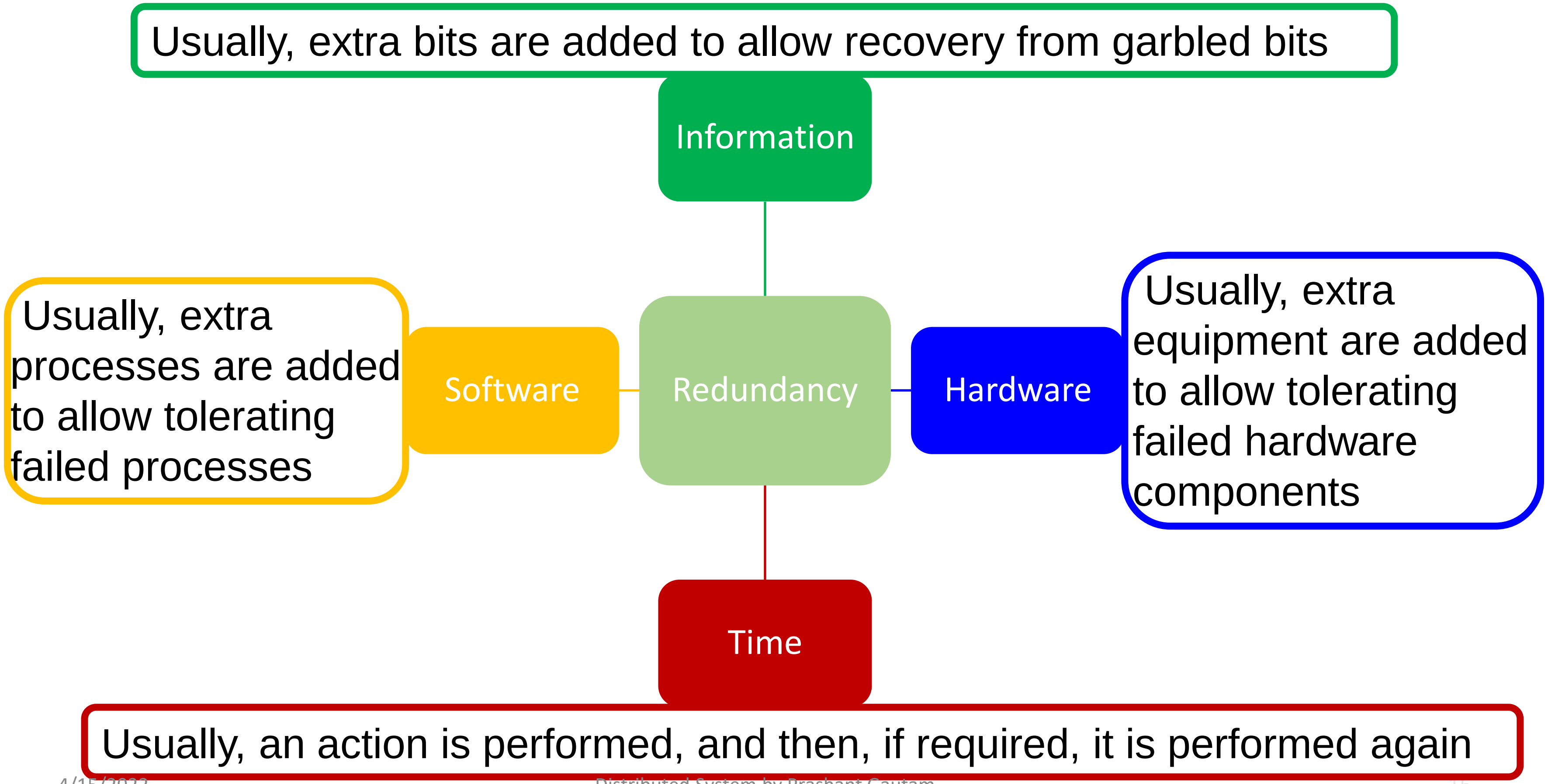
Type of Failure	Description
• Crash Failure	• A server halts, but was working correctly until it stopped
• Omission Failure • Receive Omission • Send Omission	• A server fails to respond to incoming requests • A server fails to receive incoming messages • A server fails to send messages
• Timing Failure	• 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

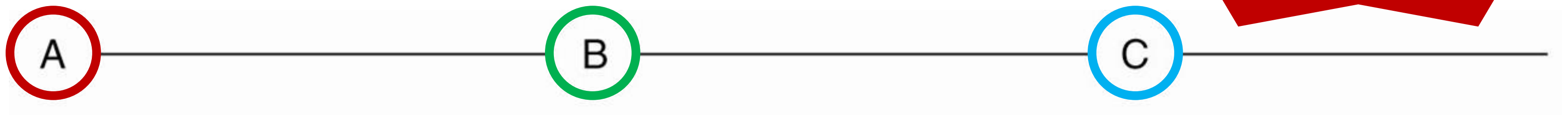
Faults Masking by Redundancy

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


Usually, extra bits are added to allow recovery from garbled bits
Information
Usually, extra processes are added to allow tolerating failed processes
Software
Redundancy
Hardware
Usually, extra equipment are added to allow tolerating failed hardware components
Time
Usually, an action is performed, and then, if required, it is performed again

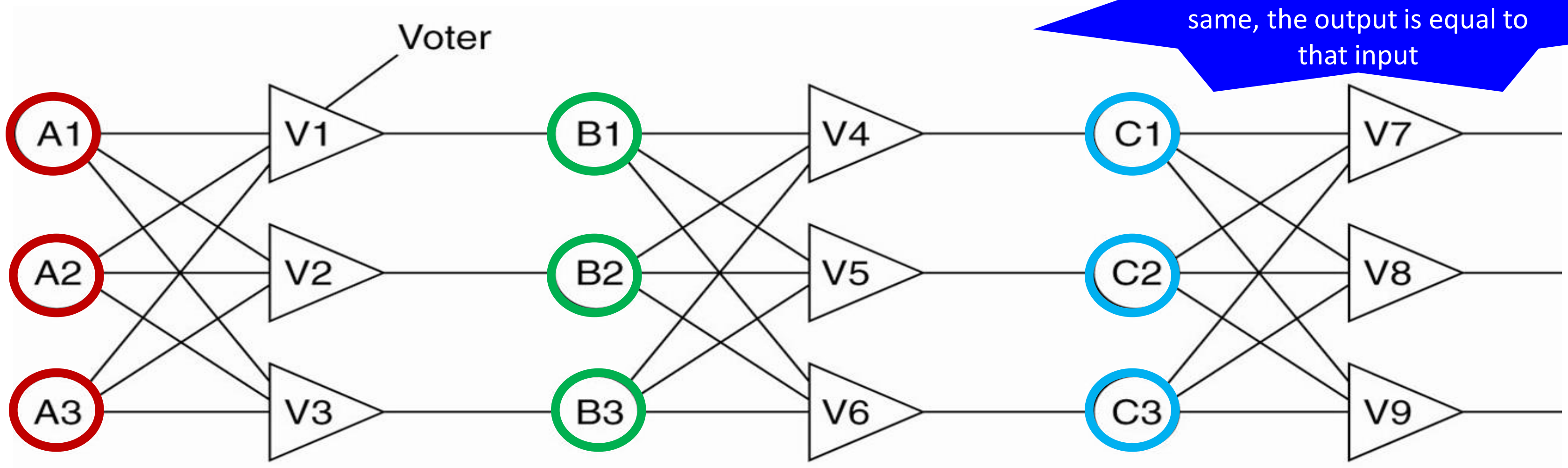
Triple Modular Redundancy

If one is faulty, the final result will be incorrect



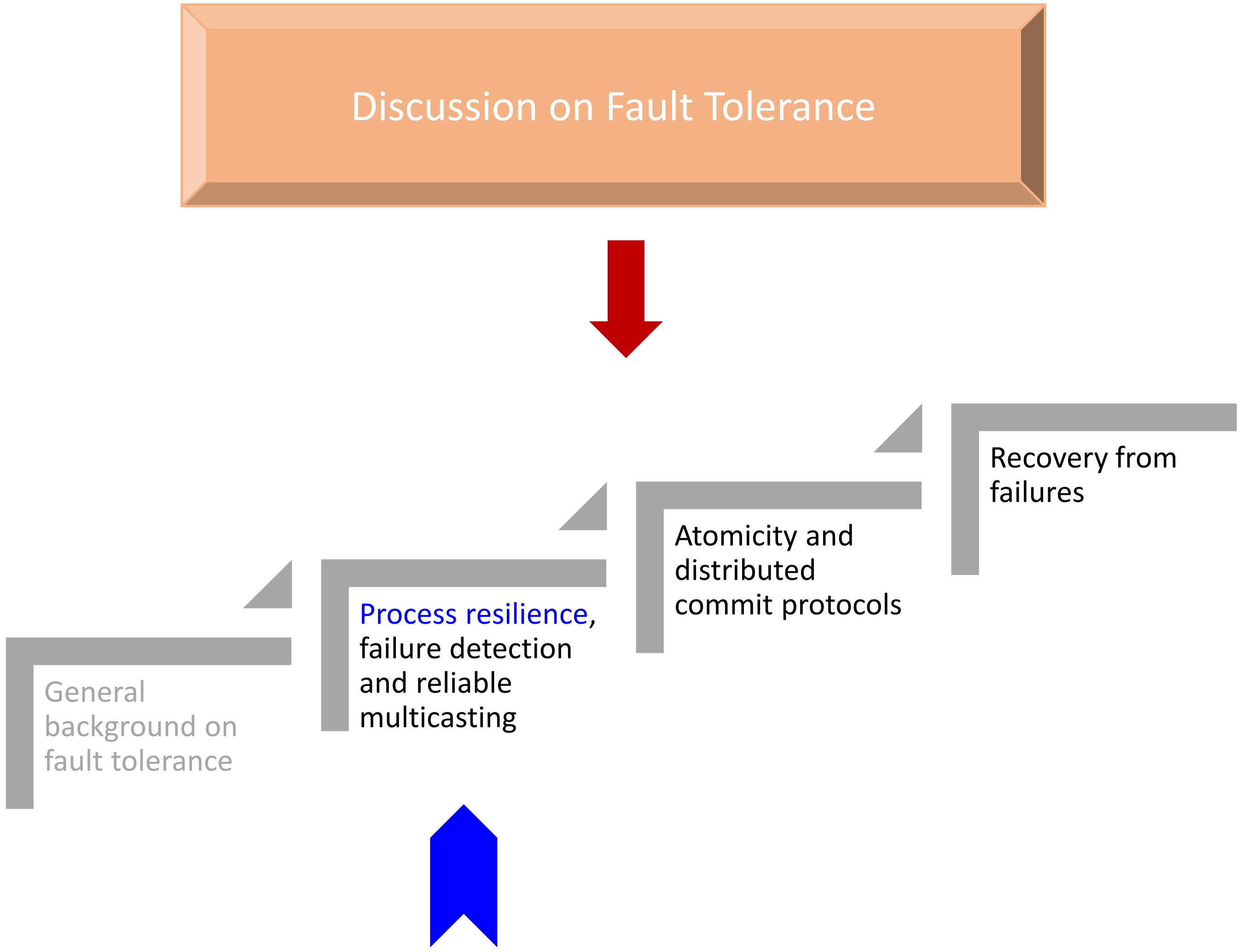
A circuit with signals passing through devices A, B, and C, in sequence

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

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

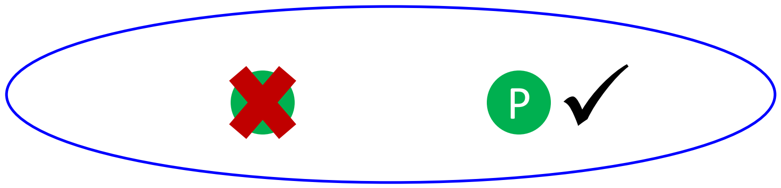
Process Resilience and Failure Detection

- 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

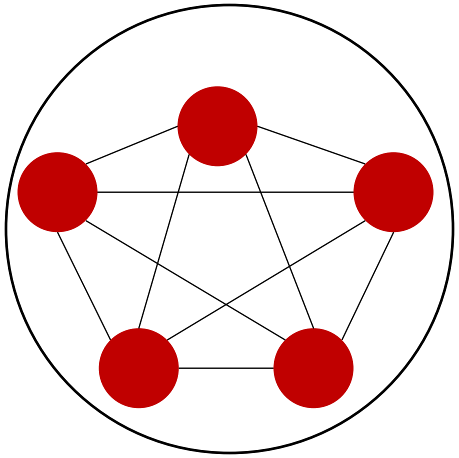
- The key approach to tolerating a faulty process is to organize several identical processes into a ***group***

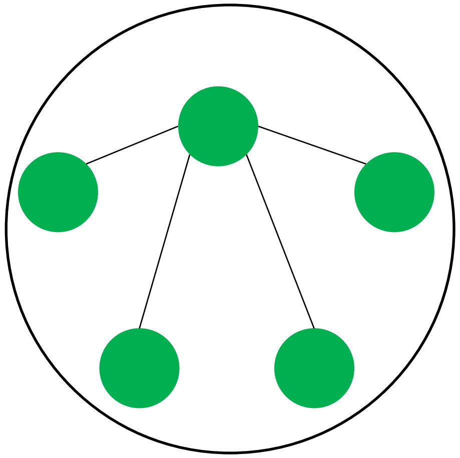


- If one process in a group fails, hopefully some other process can take over
- Caveats:
  - A process can join a group or leave one during system operation
  - A process can be a member of several groups at the same time

Flat Versus Hierarchical Groups

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





Flat Group:

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

Hierarchical Group:

(+) 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)

- 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
				✓	✓	Unbounded
	Asynchronous				✓	Bounded
					✓	Unbounded
		Unicast	Multicast	Unicast	Multicast	Communication Delay
		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* [Lamport *et al.*]

Byzantine Agreement Problem (1)

- Lamport assumes:
  - 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 v_i to the others
  - There are at most *k* faulty processes

Byzantine Agreement Problem (2)

❑ Lamport's Assumptions:

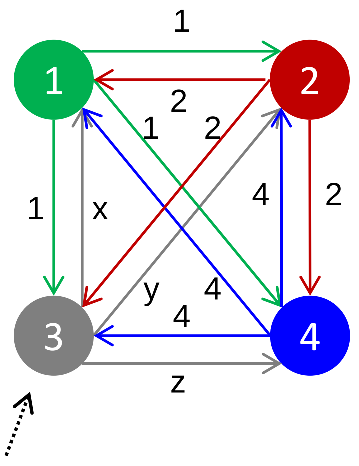
		Message Ordering				
		Unordered		Ordered		
Process Behavior	Synchronous	✓	✓	✓	✓	Bounded
				✓	✓	Unbounded
	Asynchronous				✓	Bounded
					✓	Unbounded
		Unicast	Multicast	Unicast	Multicast	Communication Delay
		Message Transmission				

Lamport suggests that each process *i* constructs a vector V of length N, such that if process *i* is non-faulty, V[*i*] = v*i*. Otherwise, V[*i*] is undefined

Byzantine Agreement Problem (3)

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

2 Got

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

4 Got

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

Byzantine Agreement Problem (4)

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

The algorithm reaches an agreement

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

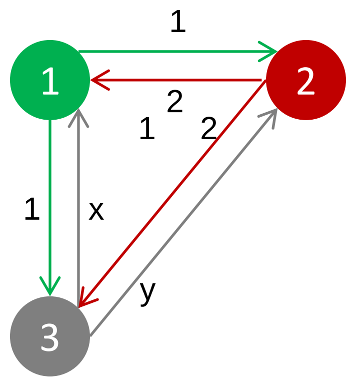
Result Vector:
(1, 2, UNKNOWN, 4)

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

Byzantine Agreement Problem (5)

- **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	**2** Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

Byzantine Agreement Problem (6)

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, y)
(a, b, c)

The algorithm has failed to produce an agreement

2 Got

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

Result Vector:
(UNKOWN, UNKNOWN, UNKNOWN)

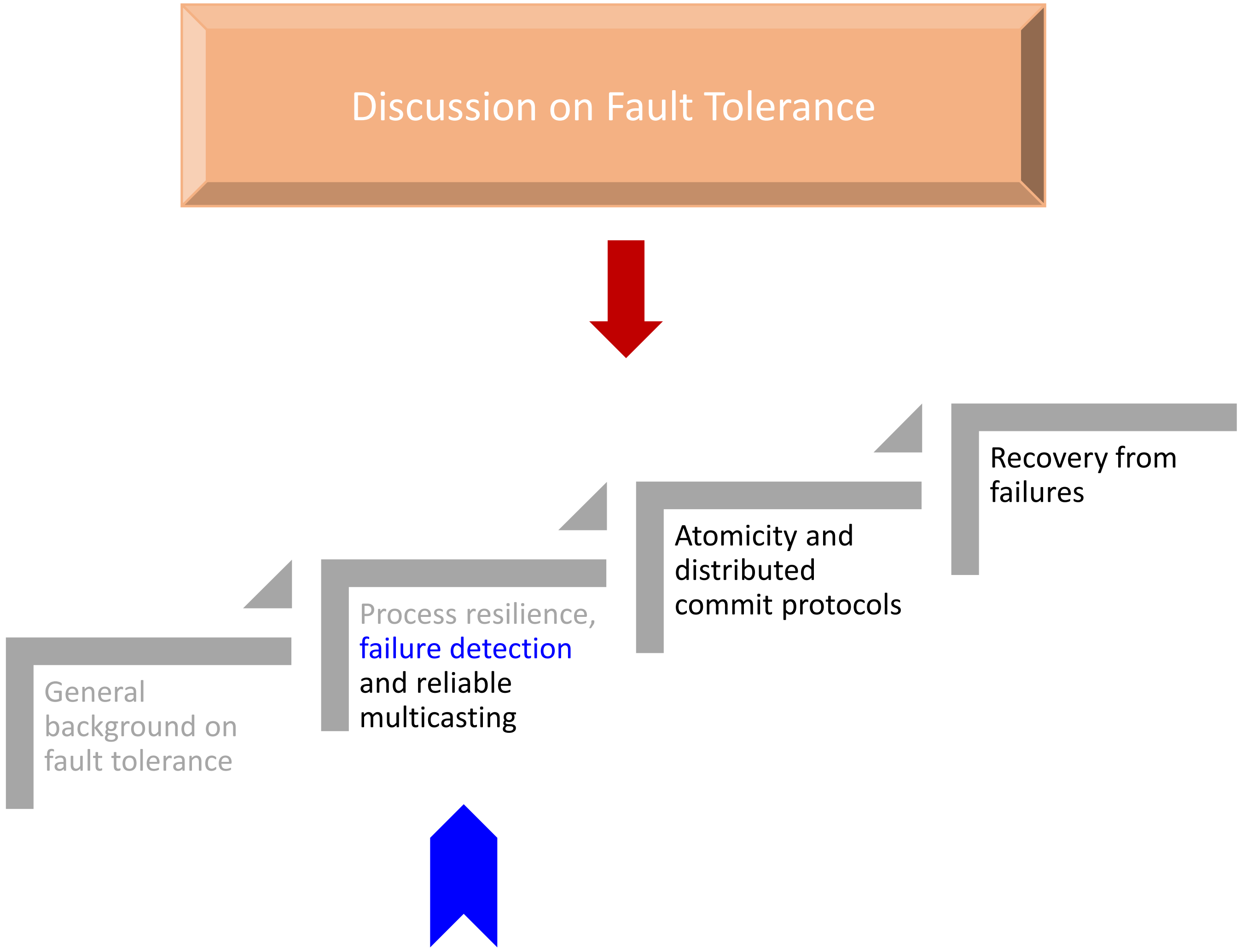
Result Vector:
(UNKOWN, UNKNOWN, UNKNOWN)

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.

- *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, <u>no agreement</u> is possible even if only one process is faulty.

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

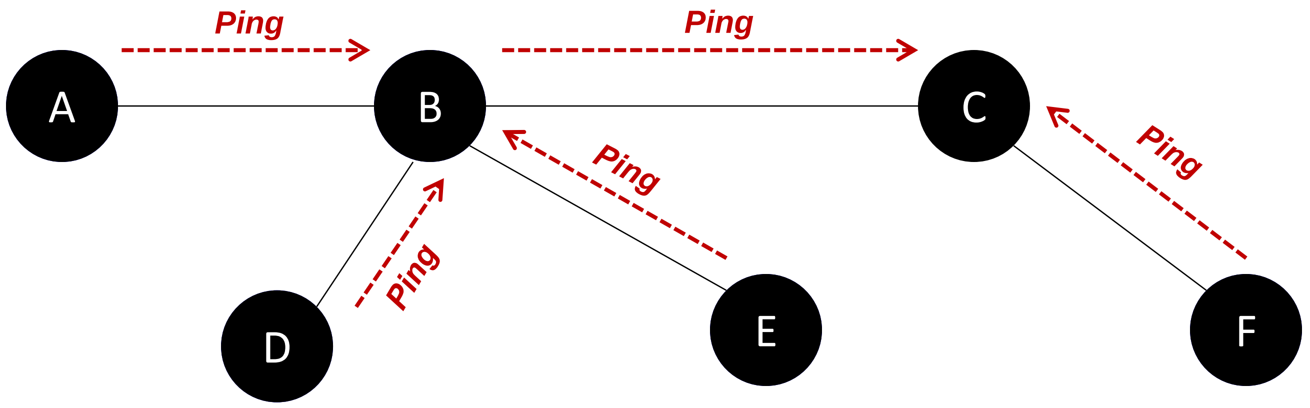
- Before we properly mask failures, we generally need to detect them
- For a group of processes, non-faulty members should be able to decide who is still a member and who is not
- Two policies:
  - Processes actively send “are you alive?” messages to each other (i.e., *pinging each other*)
  - Processes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

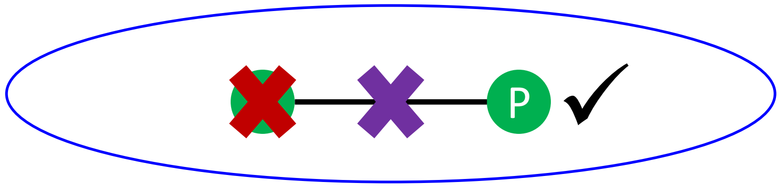
1. Failure detection can be done as a side-effect of regularly exchanging information with neighbors (e.g., ***gossip-based information dissemination***)

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

Reliable Communication

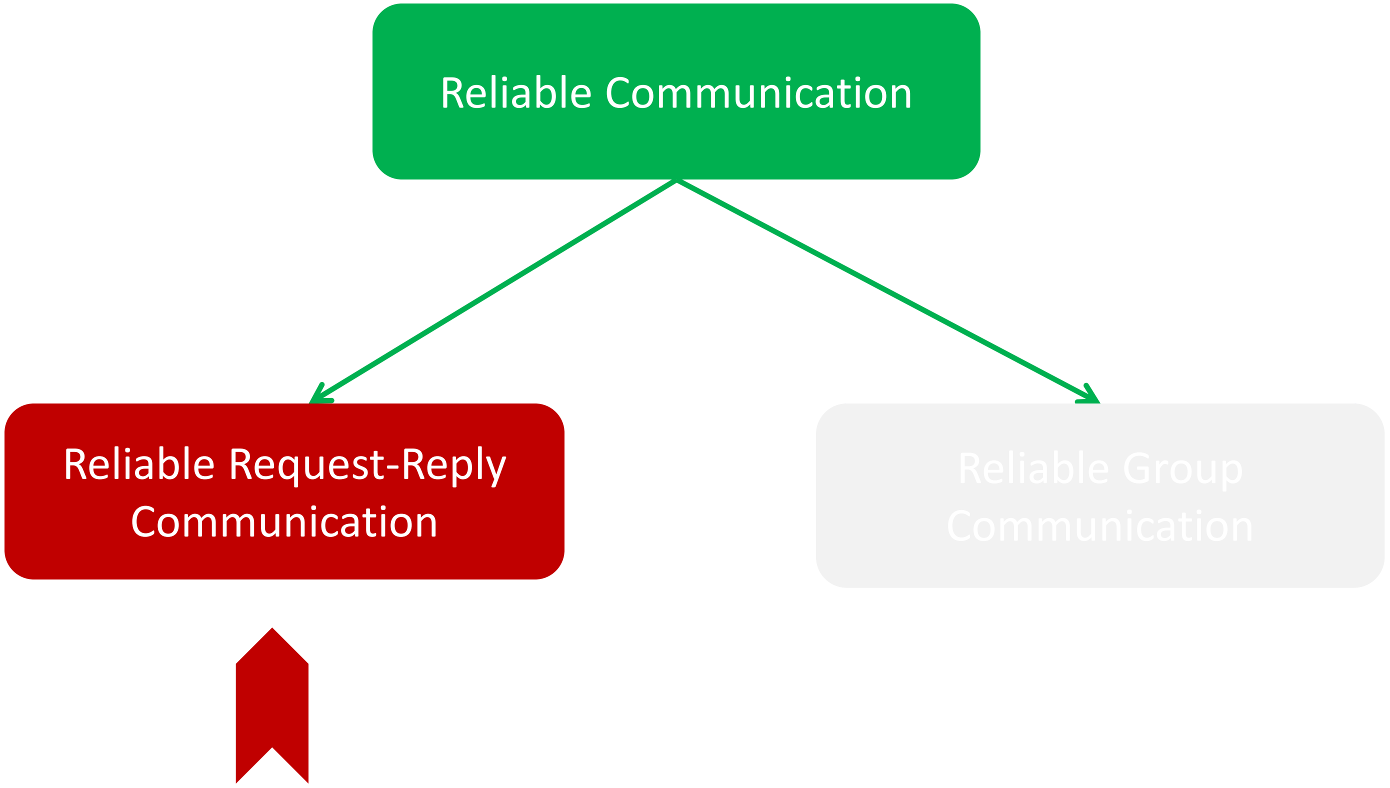
- Fault tolerance in distributed systems typically concentrates on faulty processes



- However, we also need to consider *communication failures*

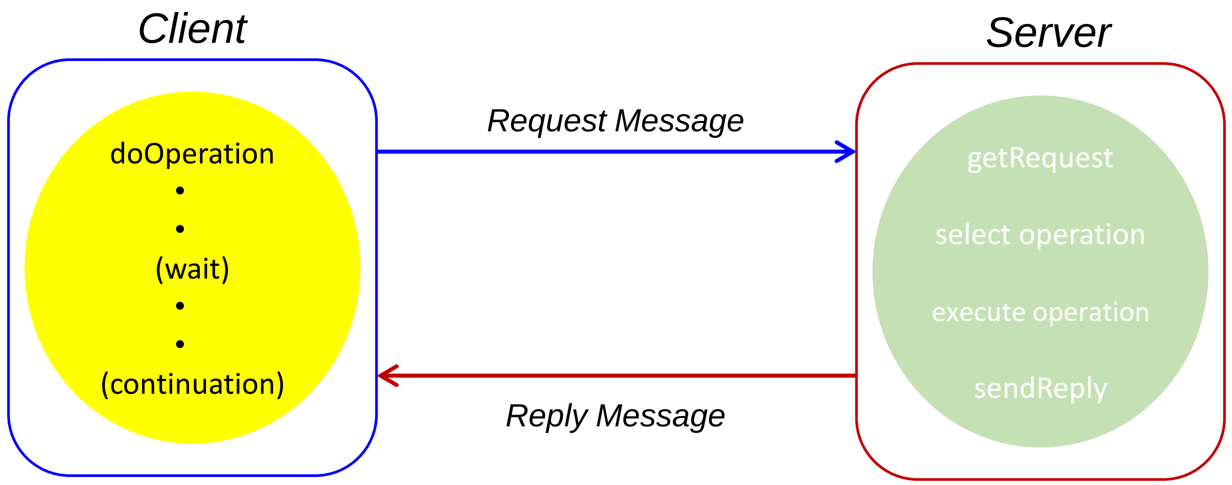
- We will focus on two types of reliable communication:
  - Reliable request-reply communication (e.g., RPC)
  - Reliable group communication (e.g., multicasting schemes)

Reliable Communication


Reliable Communication
Reliable Request-Reply Communication
Reliable Group Communication

Request-Reply Communication

- The request-reply communication is designed to support the roles and message exchanges in typical client-server interactions



- This sort of communication is mainly based on a trio of communication primitives, *doOperation*, *getRequest* and *sendReply*

Timeouts

- Request-reply communication may suffer from *crash*, *omission*, *timing*, and *byzantine* failures
- To allow for occasions where a request or a reply message is not delivered (e.g., lost), *doOperation* uses a ***timeout***
- There are various options as to what *doOperation* can do after a timeout:
  - Return immediately with an indication to the client that the request has failed
  - Send the request message repeatedly until either a reply is received or the server is assumed to have failed

Duplicate Filtering

- In cases when the request message is retransmitted, the server may receive it more than once

- This can cause the server executing an operation more than once for the same request

- To avoid this, the server should recognize successive messages from the same client and *filter out duplicates*

Lost Reply Messages

- If the server has already sent the reply when it receives a duplicate request, it can either:
  - Re-execute the operation again to obtain the result
  - Or do not re-execute the operation if it has chosen to retain the outcome of the first execution
- Not every operation can be executed more than once and obtain the same results each time
- An *idempotent* operation is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once

History

- For servers that require retransmission of replies without re-execution of operations, a *history* may be used

- The term ‘history’ is used to refer to a structure that contains a record of (reply) messages that have been transmitted

Fields of a history record:

Request ID	Message	Client ID

History

- The server can interpret each request from a client as an ACK of its previous reply
    - Thus, the history needs contain only the last reply message sent to each client
- But, if the number of clients is large, memory cost might become a problem
- Messages in a history are normally discarded after a limited period of time

Summary of Request-Reply Protocols

- The *doOperation* can be implemented in different ways to provide different delivery guarantees. The main choices are:

  1. Retry request message: Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed

  2. Duplicate filtering: Controls when retransmissions are used and whether to filter out duplicate requests at the server

  3. Retransmission of results: Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server

Request-Reply Call Semantics

- Combinations of request-reply protocols lead to a variety of possible semantics for the reliability of remote invocations

Fault Tolerance Measure			Call Semantics
Retransmit Request Message	**Duplicate Filtering**	**Re-execute Procedure or Retransmit Reply**	
No	N/A	N/A	***Maybe***
Yes	No	Re-execute Procedure	***At-least-once***
Yes	Yes	Retransmit Reply	***At-most-once***

Maybe Semantics

- With *maybe semantics*, the remote procedure call may be executed *once or not at all*

- *Maybe semantics* arises when no fault-tolerance measures are applied and can suffer from the following types of failure:

    - Omission failures if the request or result message is lost
    - Crash failures when the server containing the remote operation fails

- *Maybe semantics* is useful only for applications in which occasional failed calls are acceptable

Maybe Semantics: Revisit

Fault Tolerance Measure			Call Semantics
Retransmit Request Message	**Duplicate Filtering**	**Re-execute Procedure or Retransmit Reply**	
No	N/A	N/A	***Maybe***
Yes	No	Re-execute Procedure	***At-least-once***
Yes	Yes	Retransmit Reply	***At-most-once***

At-Least-Once Semantics

- With at-least-once semantics, the invoker keeps *retransmitting the request message until a reply is received*
- At-least-once semantics:
  - Masks the omission failures due to retransmissions
  - Suffers from crash failures when the server containing the remote operation fails
  - Might suffer from response failures if a re-executed operation is not idempotent

At-Least-Once Semantics: Revisit

Fault Tolerance Measure			Call Semantics
Retransmit Request Message	**Duplicate Filtering**	**Re-execute Procedure or Retransmit Reply**	
No	N/A	N/A	***Maybe***
Yes	No	Re-execute Procedure	***At-least-once***
Yes	Yes	Retransmit Reply	***At-most-once***

At-Most-Once Semantics

- With at-most-once semantics, the invoker *gives up immediately and reports back a failure*

- At-most-once semantics:
  - Masks the omission failures of the request or result messages by retransmitting request messages
  - Avoids response failures by ensuring that each operation is never executed more than once

At-Most-Once Semantics: Revisit

Fault Tolerance Measure			**Call Semantics**
Retransmit Request Message	**Duplicate Filtering**	**Re-execute Procedure or Retransmit Reply**	
No	N/A	N/A	***Maybe***
Yes	No	Re-execute Procedure	***At-least-once***
Yes	Yes	Retransmit Reply	***At-most-once***

Classes of Failures in Request-Reply Communication

- There are 5 different classes of failures that can occur in request-reply systems:

1. The client is unable to locate the server
2. The request message from the client to the server is lost
3. The server crashes after receiving a request
4. The reply message from the server to the client is lost
5. The client crashes after sending a request

Classes of Failures in Request-Reply Communication

- There are 5 different classes of failures that can occur in request-reply systems:

  1. The client is unable to locate the server
  2. The request message from the client to the server is lost
  3. The server crashes after receiving a request
  4. The reply message from the server to the client is lost
  5. The client crashes after sending a request

Possible Solution

- One possible solution for the client being unable to locate the server is to have *doOperation* raise an ***exception*** at the client side
- Considerations:
  - Not every language has exceptions or signals
  - Writing an exception identifies the location of the error and hence destroys the transparency of the distributed system

Classes of Failures in Request-Reply Communication

- There are 5 different classes of failures that can occur in request-reply systems:

1. The client is unable to locate the server
2. The request message from the client to the server is lost
3. The server crashes after receiving a request
4. The reply message from the server to the client is lost
5. The client crashes after sending a request

Possible Solution

- The *doOperation* can start a *timer* when sending the request message
- If the timer expires before a reply or an ACK comes back, the message is sent again
- Considerations:
  - If the message was lost, the server might not be able to recognize the difference between a first transmission and a retransmission
  - If the message was not lost, the server has to detect that it is dealing with a retransmission request

Classes of Failures in Request-Reply Communication

- There are 5 different classes of failures that can occur in request-reply systems:

1. The client is unable to locate the server
2. The request message from the client to the server is lost
3. The server crashes after receiving a request
4. The reply message from the server to the client is lost
5. The client crashes after sending a request

Possible Solution (1)

- The *doOperation* can start a *timer* when sending the request message

- If the timer expires before a reply or an ACK comes back, the message is sent again

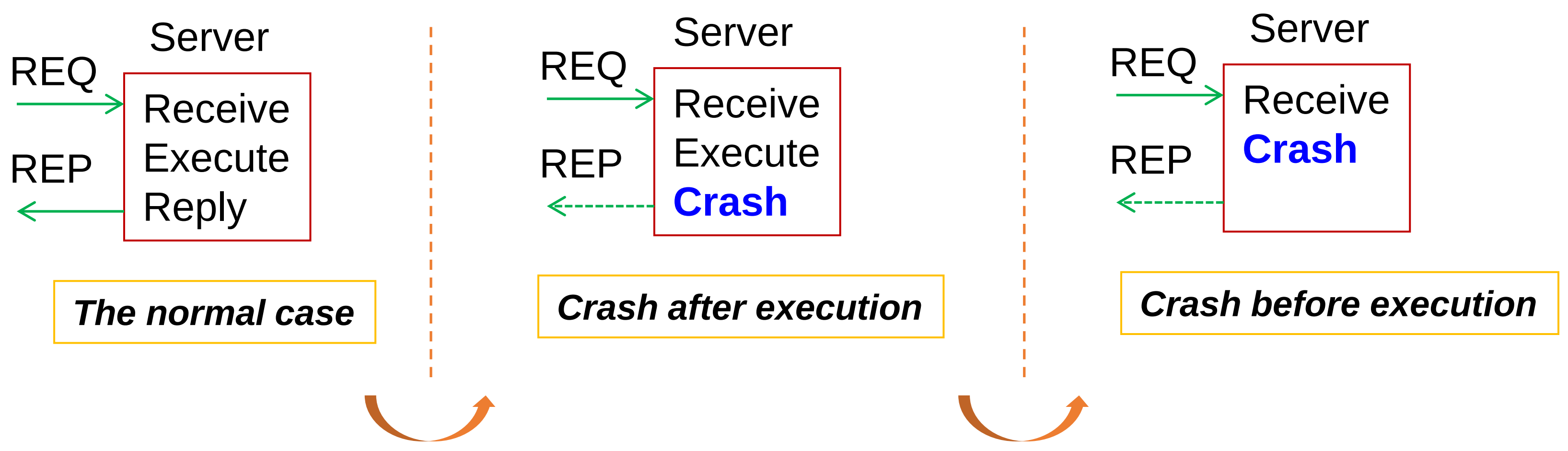
- *We can apply any of the 3 request-reply call semantics*

- Considerations:

  - The crash failure may occur *either before or after* the operation at the server is executed. The *doOperation* cannot figure that out

Possible Solution (2)

- Considerations (*Cont'd*):
  - The sequence of events at server is as follows:



  - In the last 2 cases, *doOperation* cannot tell which is which. All it knows is that its *timer* has expired

A Printing Example (PE): Normal Scenario

- A client’s remote operation consists of printing some text at a server

- When a client issues a request, it receives an ACK that the request has been delivered to the server

- The server sends a completion message to the client when the text is printed

PE: Possible Events at Server

- Three events can happen at the server:

  1. Send the completion message (M)
  2. Print the text (P)
  3. Crash (C)

PE: Server Strategies

- The server has a choice between two strategies:

  1. Send a completion message just before commanding the printer to do its work

  2. Send a completion message after the text is printed

PE: Failure Scenario

The server crashes, subsequently recovers and announces that to all clients

PE: Server Events Ordering

- Server events can occur in six different orderings:

Ordering	Description
M→P→C	A crash occurs after sending the completion message and printing the text
M→C(→P)	A crash occurs after sending the completion message, but before the text is printed
P→M→C	A crash occurs after printing the text and sending the completion message
P→C(→M)	The text is printed, after which a crash occurs before the completion message is sent
C(→P→M)	A crash happens before the server could do anything
C(→M→P)	A crash happens before the server could do anything

PE: Client Reissue Strategies

- After the crash of the server, the client does not know whether its request to print some text was carried out or not

- The client has a choice between 4 strategies:

Reissue Strategy	Description
Never	Never reissue a request, at the risk that the text will not be printed
Always	Always reissue a request, potentially leading to the text being printed twice
Reissue When Not ACKed	Reissue a request only if it did not yet receive an ACK that its print request had been delivered to the server
Reissue When ACKed	Reissue a request only if it has received an ACK for the print request

PE: Summary and Conclusion

- In summary, the different combinations of client and server strategies are as follows *(OK= Text is printed once, DUP= Text is printed twice, and ZERO= Text is not printed)*:

▪There is NO combination of a client strategy and a server strategy that will work correctly under all possible event sequences

▪The client can never know whether the server crashed just before or after having the text printed

Classes of Failures in Request-Reply Communication

- There are 5 different classes of failures that can occur in request-reply systems:

1. The client is unable to locate the server
2. The request message from the client to the server is lost
3. The server crashes after receiving a request
4. The reply message from the server to the client is lost
5. The client crashes after sending a request

Possible Solution (1)

- The *doOperation* can start a *timer* when sending the request message
- If the timer expires before a reply or an ACK comes back, the message is sent again
- Considerations:
  - For that to happen, the client's request should be *idempotent*

Possible Solution (2)

- What if the client's request is not idempotent?
  - Have the client assign each request a sequence number
  - Have the server keep track of the most recently received sequence number from each client
  - The server can then tell the difference between an original request and a retransmission one and can refuse to carry out any request a second time

Classes of Failures in Request-Reply Communication

- There are 5 different classes of failures that can occur in request-reply systems:

1. The client is unable to locate the server
2. The request message from the client to the server is lost
3. The server crashes after receiving a request
4. The reply message from the server to the client is lost
5. The client crashes after sending a request

Orphans

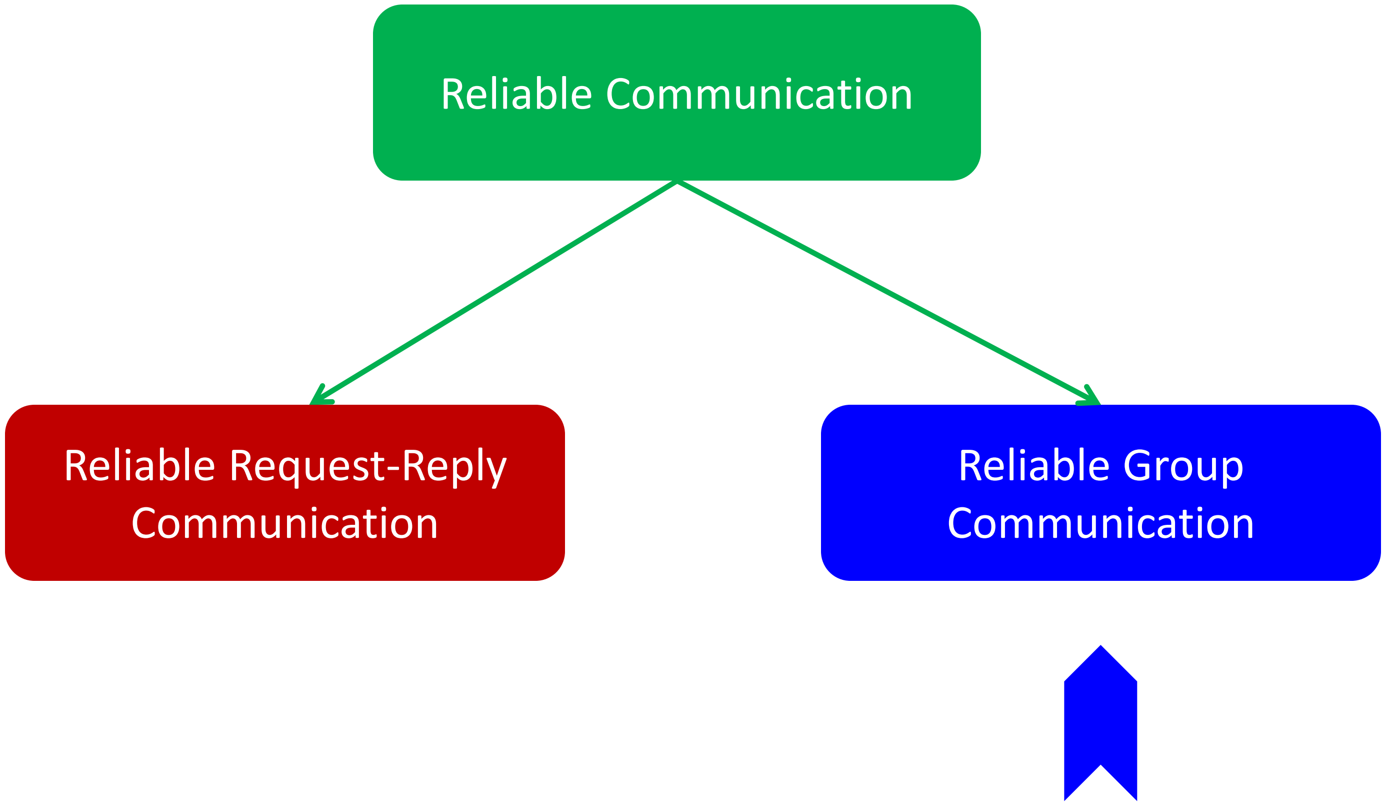
- A client might crash while the server is performing a corresponding computation
  - Such an unwanted computation is called an *orphan* (as there is no parent waiting for it after done)
- Orphans can cause a variety of problems that can interfere with the normal operation of the system:
  - They waste CPU cycles
  - They might lock up files and tie up valuable resources
  - If the client reboots, does the request again, and then an orphan reply comes back immediately afterwards, a confusion might occur

Possible Solutions [Nelson 1981]

- ***S1: Extermination***: Use logging to explicitly kill off an orphan after a client reboot
- ***S2: Reincarnation:*** Use broadcasting to kill *all* remote computations on a client's behalf after rebooting and getting a new *epoch number*
- ***S3: Gentle Reincarnation***: After an epoch broadcast comes in, a machine kills a computation only if its owner cannot be located anywhere
- ***S4: Expiration***: each remote invocation is given a standard amount of time to fulfill the job

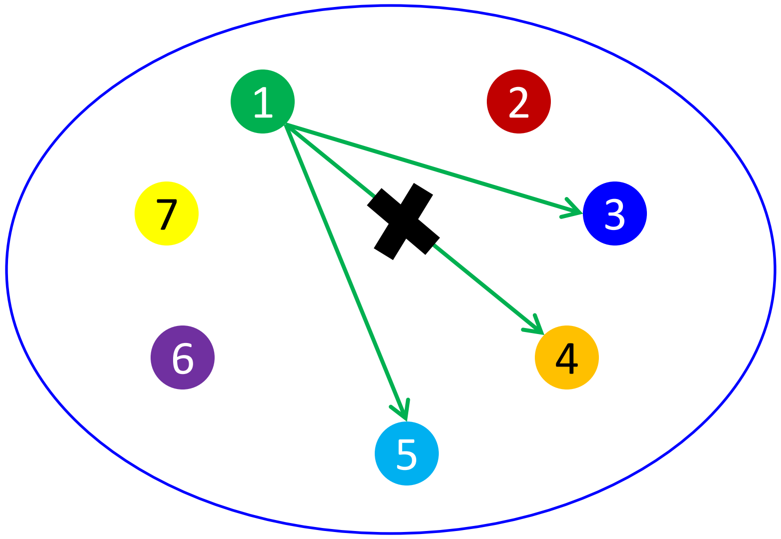
Orphan elimination is discussed in more detail by ***Panzieri and Shrivastave (1988)***

Reliable Communication


Reliable Communication
Reliable Request-Reply Communication
Reliable Group Communication

Reliable Group Communication

- As we considered reliable request-reply communication, we need also to consider reliable multicasting services



- E.g., Election algorithms use multicasting schemes

Reliable Group Communication

- A Basic Reliable-Multicasting Scheme
- Scalability in Reliable Multicasting
- Atomic Multicast

Reliable Group Communication

- A Basic Reliable-Multicasting Scheme
- Scalability in Reliable Multicasting
- Atomic Multicast

Reliable Multicasting

- Reliable multicasting indicates that a message that is sent to a process group should be delivered to each member of that group

- A distinction should be made between:

  - Reliable communication in the presence of faulty processes
  - Reliable communication when processes are assumed to operate correctly

- In the presence of faulty processes, multicasting is considered to be reliable when it can be guaranteed that *all non-faulty* group members receive the message

Basic Reliable Multicasting Questions

- What happens if during communication (i.e., a message is being delivered) a process *P* joins a group?
  - Should *P* also receive the message?
- What happens if a (sending) process crashes during communication?
- What about message ordering?

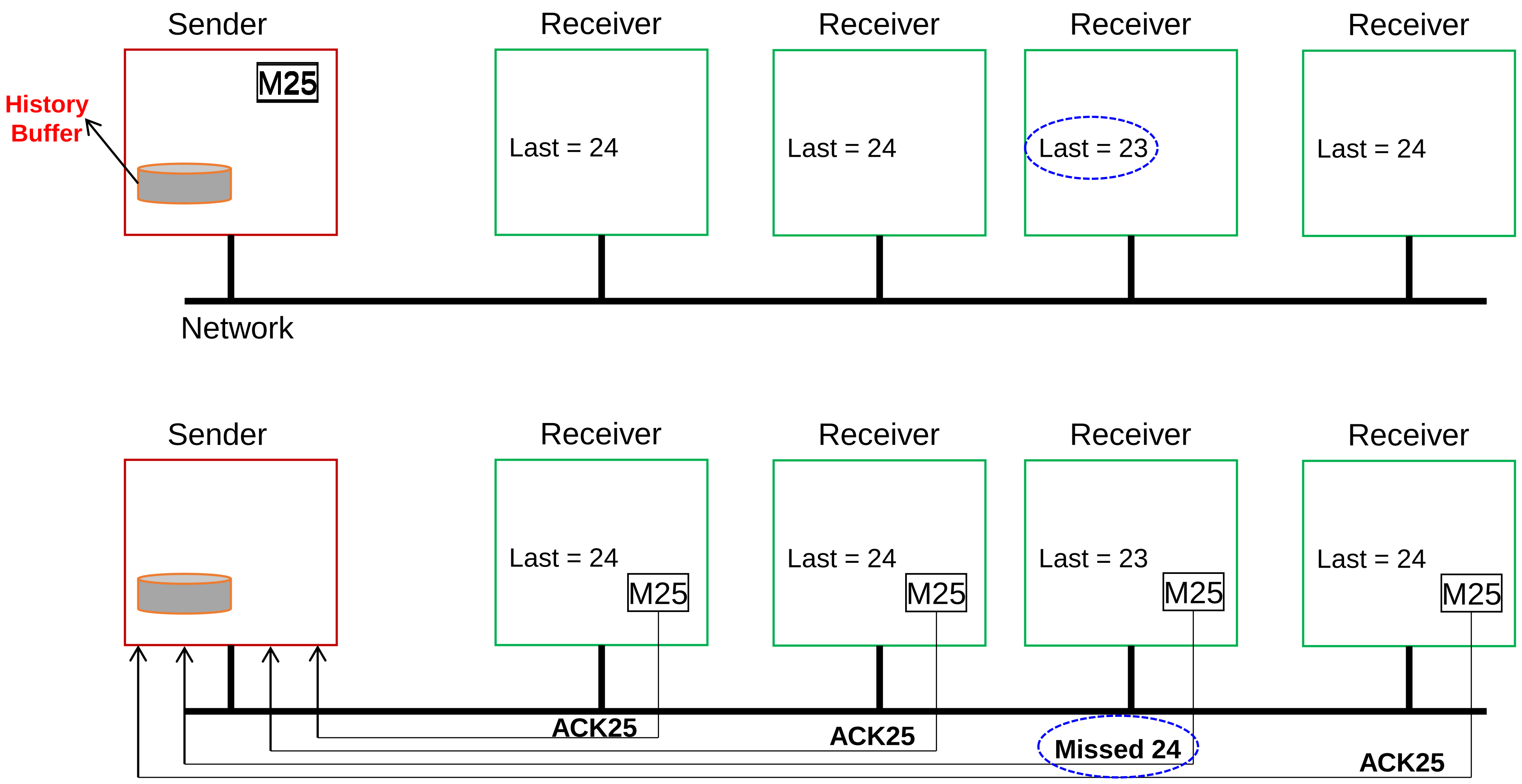
Reliable Multicasting with Feedback Messages

- Consider the case when a single sender ***S*** wants to multicast a message to multiple receivers

- An ***S's*** multicast message may be lost part way and delivered to some, but not to all, of the intended receivers

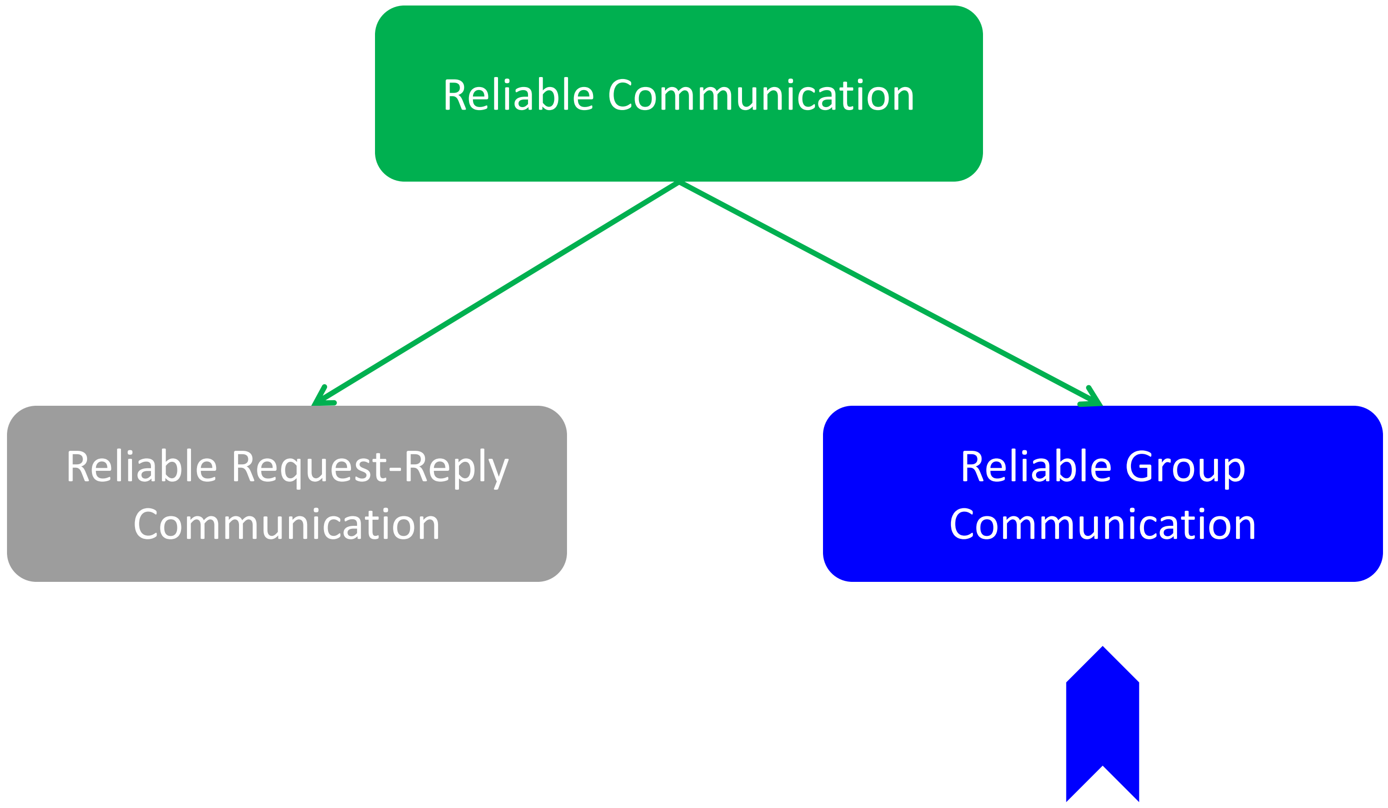
- Assume that messages are received in the same order as they are sent

Reliable Multicasting with Feedback Messages



An extensive and detailed survey of total-order broadcasts can be found in Defago et al. (2004)

Reliable Communication


Reliable Communication
Reliable Request-Reply Communication
Reliable Group Communication

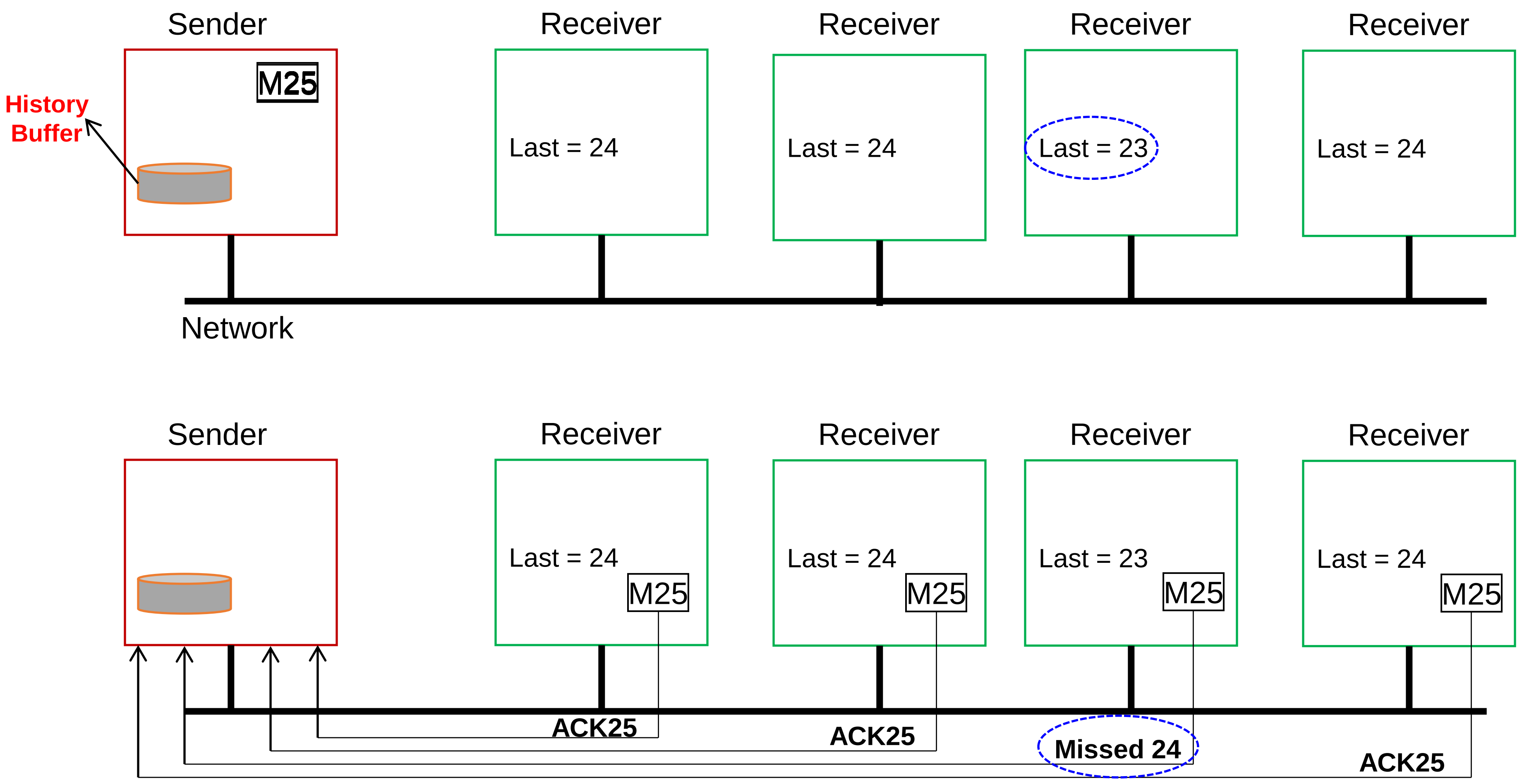
Reliable Group Communication

- A Basic Reliable-Multicasting Scheme
- Scalability in Reliable Multicasting
- Atomic Multicast

Reliable Group Communication

- A Basic Reliable-Multicasting Scheme
- Scalability in Reliable Multicasting
- Atomic Multicast

Reliable Multicasting with Feedback Messages



Reliable Group Communication

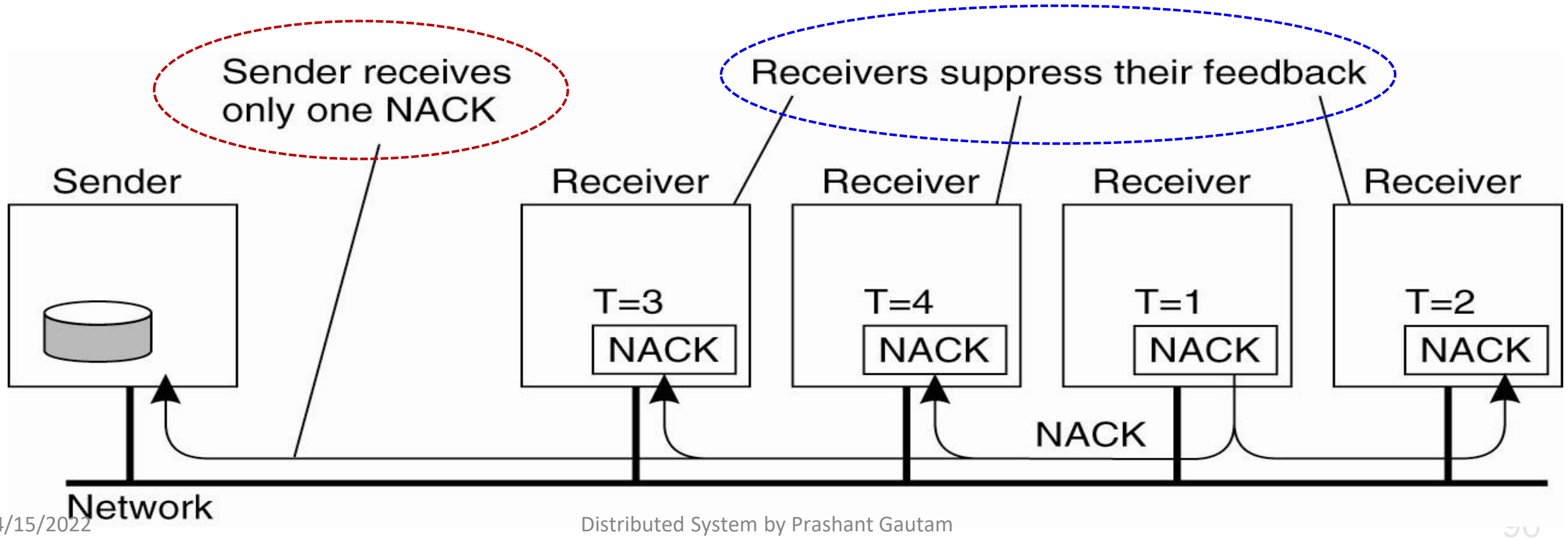
- A Basic Reliable-Multicasting Scheme
- Scalability in Reliable Multicasting
- Atomic Multicast

Scalability Issues with a Feedback-Based Scheme

- If there are *N* receivers in a multicasting process, the sender must be prepared to accept at least *N* ACKs
- This might cause a *feedback implosion*
- Instead, we can let a receiver return only a Negative ACK
- Limitations:
  - No hard guarantees can be given that a feedback implosion will not happen
  - It is not clear for how long the sender should keep a message in its history buffer

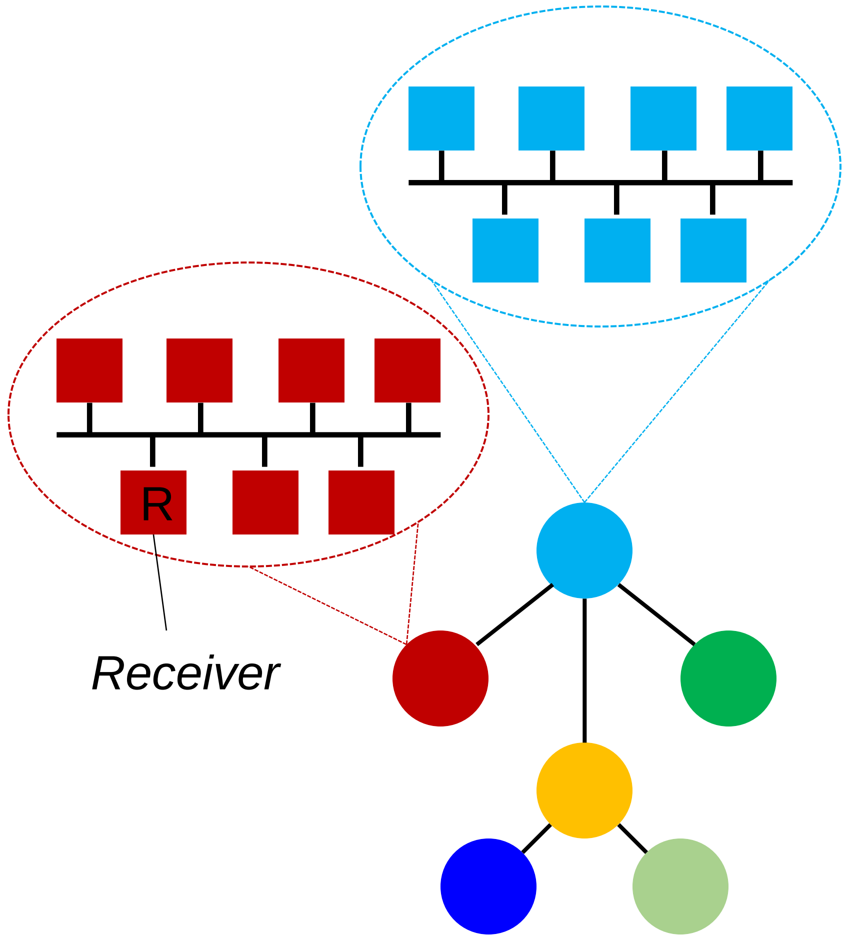
Nonhierarchical Feedback Control

- *How can we control the number of NACKs sent back to the sender?*
  - A NACK is sent to all the group members after some *random delay*
  - A group member *suppresses* its own feedback concerning a missing message after receiving a NACK feedback about the same message



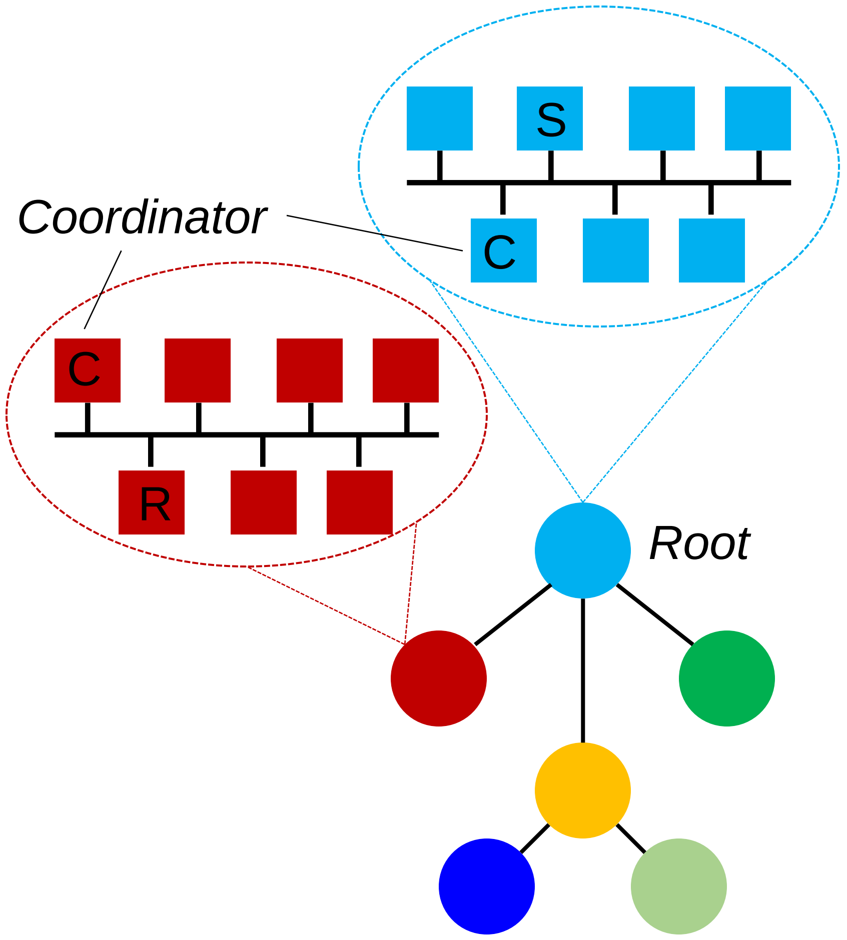
Hierarchical Feedback Control

- Feedback suppression is basically a nonhierarchical solution
- Achieving scalability for very large groups of receivers requires that hierarchical approaches are adopted
- The group of receivers is partitioned into a number of subgroups, which are organized into a *tree*



Hierarchical Feedback Control

- The subgroup containing the sender *S* forms the *root* of the tree
- Within a subgroup, any reliable multicasting scheme can be used
- Each subgroup appoints a local *coordinator C* responsible for handling retransmission requests in its subgroup
- If *C* misses a message *m,* it asks the *C* of the parent subgroup to retransmit *m*



Reliable Group Communication

- A Basic Reliable-Multicasting Scheme
- Scalability in Reliable Multicasting
- Atomic Multicast

Atomic Multicast

- *P1*: What is often needed in a distributed system is the guarantee that a message is delivered *to either all processes or to none at all*

- *P2*: It is also generally required that all messages are delivered *in the same order to all processes*

- Satisfying *P1* and *P2* results in an atomic multicast

- Atomic multicast:
    - Ensures that non-faulty processes maintain a *consistent view*
    - Forces reconciliation when a process recovers and rejoins the group

Distributed Commit

- Atomic multicasting problem is an example of a more general problem, known as *distributed commit*
- The distributed commit problem involves having an operation being performed by *each* member of a process group, or *none* at all
  - With reliable multicasting, the operation is the delivery of a message
  - With distributed transactions, the operation may be the commit of a transaction at a single site that takes part in the transaction
- Distributed commit is often established by means of a *coordinator* and *participants*

One-Phase Commit Protocol

- In a simple scheme, a coordinator can tell all participants whether or not to (locally) perform the operation in question
  - This scheme is referred to as a *one-phase commit protocol*
- The one-phase commit protocol has a main drawback that if one of the participants cannot actually perform the operation, there is no way to tell the coordinator
- In practice, more sophisticated schemes are needed. The most common utilized one is the *two-phase commit protocol*

Two-Phase Commit Protocol

- Assuming that no failures occur, the two-phase commit protocol (2PC) consists of the following two phases, each consisting of two steps:

Phase I: Voting Phase	
Step 1	• The coordinator sends a VOTE_REQUEST message to all participants.
Step 2	• When a participant receives a VOTE_REQUEST message, it returns either a VOTE_COMMIT message to the coordinator indicating that it is prepared to locally commit its part of the transaction, or otherwise a VOTE_ABORT message.

Two-Phase Commit Protocol

Phase II: Decision Phase	
Step 1	• The coordinator collects all votes from the participants. • If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a GLOBAL_COMMIT message to all participants. • However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a GLOBAL_ABORT message.
Step 2	• Each participant that voted for a commit waits for the final reaction by the coordinator. • If a participant receives a GLOBAL_COMMIT message, it locally commits the transaction. • Otherwise, when receiving a GLOBAL_ABORT message, the transaction is locally aborted as well.

Recovery

- So far, we have mainly concentrated on algorithms that allow us to tolerate faults
- However, once a failure has occurred, it is essential that the process where the failure has happened can *recover* to a *correct state*
- In what follows we focus on:
  - What it actually means to recover to a correct state
  - When and how the state of a distributed system can be recorded and recovered, by means of *checkpointing* and *message logging*

Recovery

- Error Recovery
- Checkpointing
- Message Logging

Recovery

- Error Recovery
- Checkpointing
- Message Logging

Error Recovery

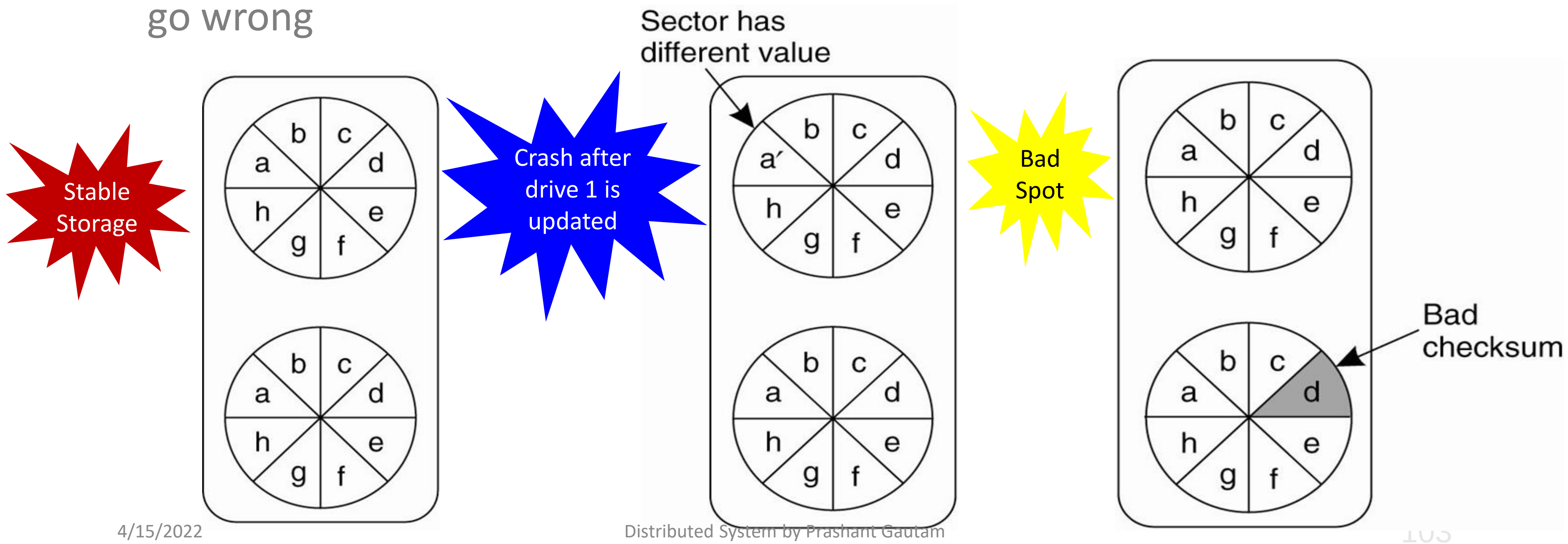
- Once a failure has occurred, it is essential that the process where the failure has happened can recover to a correct state
- Fundamental to fault tolerance is the recovery from an error
- The idea of *error recovery* is to replace an erroneous state with an error-free state
- There are essentially two forms of error recovery:

  1. *Backward recovery*
  2. *Forward recovery*

1. Backward Recovery (1)

- In backward recovery, the main issue is to bring the system from its present erroneous state back to a previously correct state

- It is necessary to record the system's state *from time to time* onto a *stable storage*, and to restore such a recorded state when things go wrong



1. Backward Recovery (2)

- Each time (part of) the system's present state is recorded, a *checkpoint* is said to be made

- Problems with backward recovery:
  - Restoring a system or a process to a previous state is generally expensive in terms of performance
  - Some states can never be rolled back (e.g., typing in UNIX rm –fr *)

2. Forward Recovery

- When the system detects that it has made an error, forward recovery reverts the system state to error time and corrects it, to be able to move forward

- Forward recovery is typically faster than backward recovery but requires that it has to be known in advance which errors may occur

- Some systems make use of both forward and backward recovery for different errors or different parts of one error

Recovery

- Error Recovery
- Checkpointing
- Message Logging

Why Checkpointing?

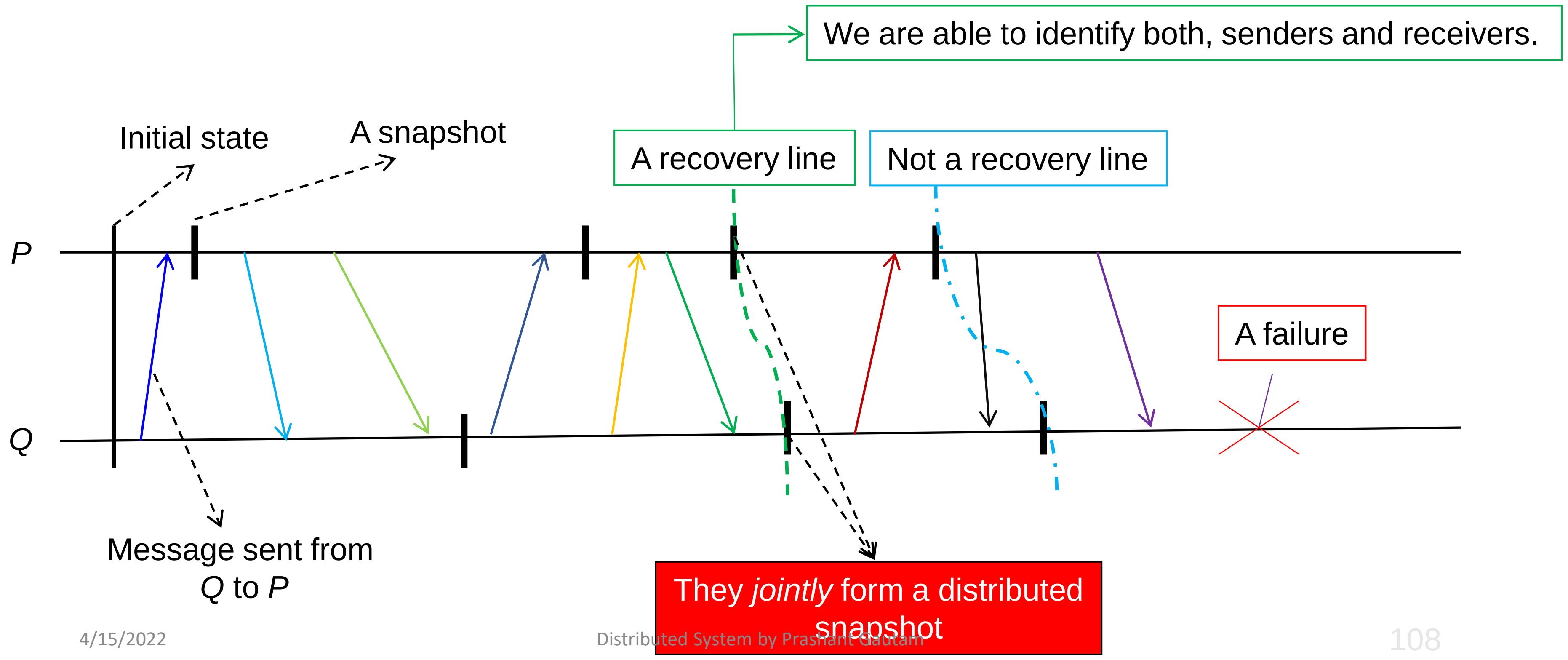
- In a fault-tolerant distributed system, backward recovery requires that the system regularly saves its state onto a stable storage

- This process is referred to as checkpointing

- In particular, checkpointing consists of storing a *distributed snapshot* of the current application state (i.e., *a consistent global state*), and later on, use it for restarting the execution in case of a failure

Recovery Line

- In a distributed snapshot, if a process *P* has recorded the receipt of a message, then there should be also a process *Q* that has recorded the sending of that message


We are able to identify both, senders and receivers.
Initial state
A snapshot
A recovery line
Not a recovery line
P
Q
A failure
Message sent from Q to P
They *jointly* form a distributed snapshot

Checkpointing

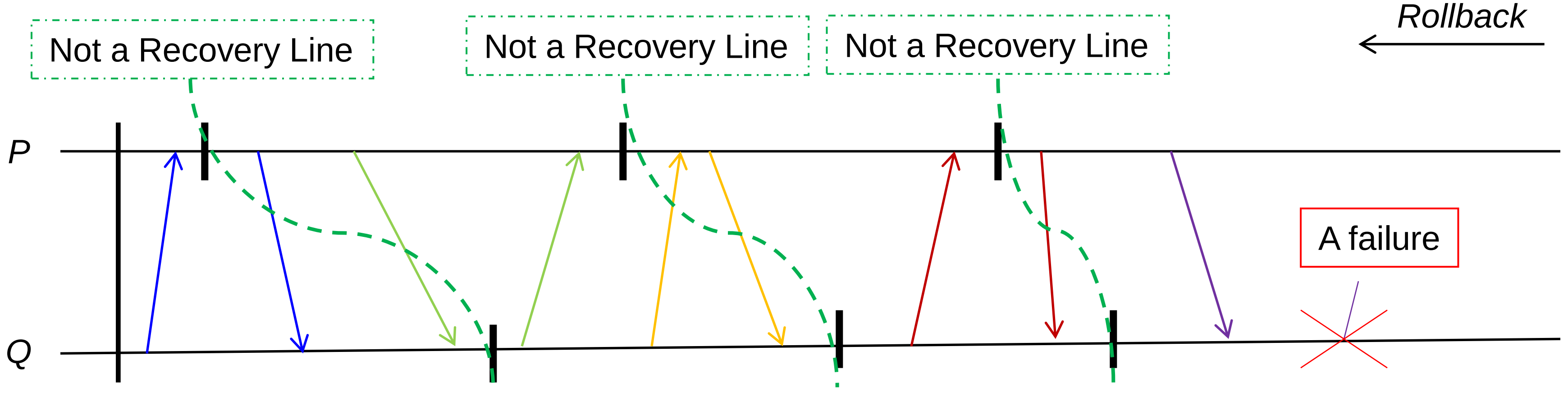
- Checkpointing can be of two types:

1. Independent Checkpointing: each process simply records its local state from time to time in an uncoordinated fashion

2. Coordinated Checkpointing: all processes synchronize to jointly write their states to local stable storages

Domino Effect

- Independent checkpointing may make it difficult to find a recovery line, leading potentially to a <u>*domino effect*</u> resulting from *cascaded* rollbacks



- With coordinated checkpointing, the saved state is automatically globally consistent, hence, domino effect is inherently avoided

Recovery

- Error Recovery
- Check-pointing
- Message Logging

Why Message Logging?

- Considering that check-pointing is an expensive operation, techniques have been sought to reduce the number of checkpoints, but still enable recovery

- An important technique in distributed systems is *message logging*

- The basic idea is that if transmission of messages can be replayed, we can still reach a globally consistent state but without having to restore that state from stable storage

- In practice, the combination of having fewer checkpoints and message logging is more efficient than having to take many checkpoints

Message Logging

- Message logging can be of two types:

  1. Sender-based logging: A process can log its messages before sending them off

  2. Receiver-based logging: A receiving process can first log an incoming message before delivering it to the application

- When a sending or a receiving process crashes, it can restore the most recently checkpointed state, and from there on *replay* the logged messages (important for non-deterministic behaviors)