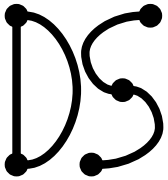


Research Group
Distributed Systems


# Distributed Systems

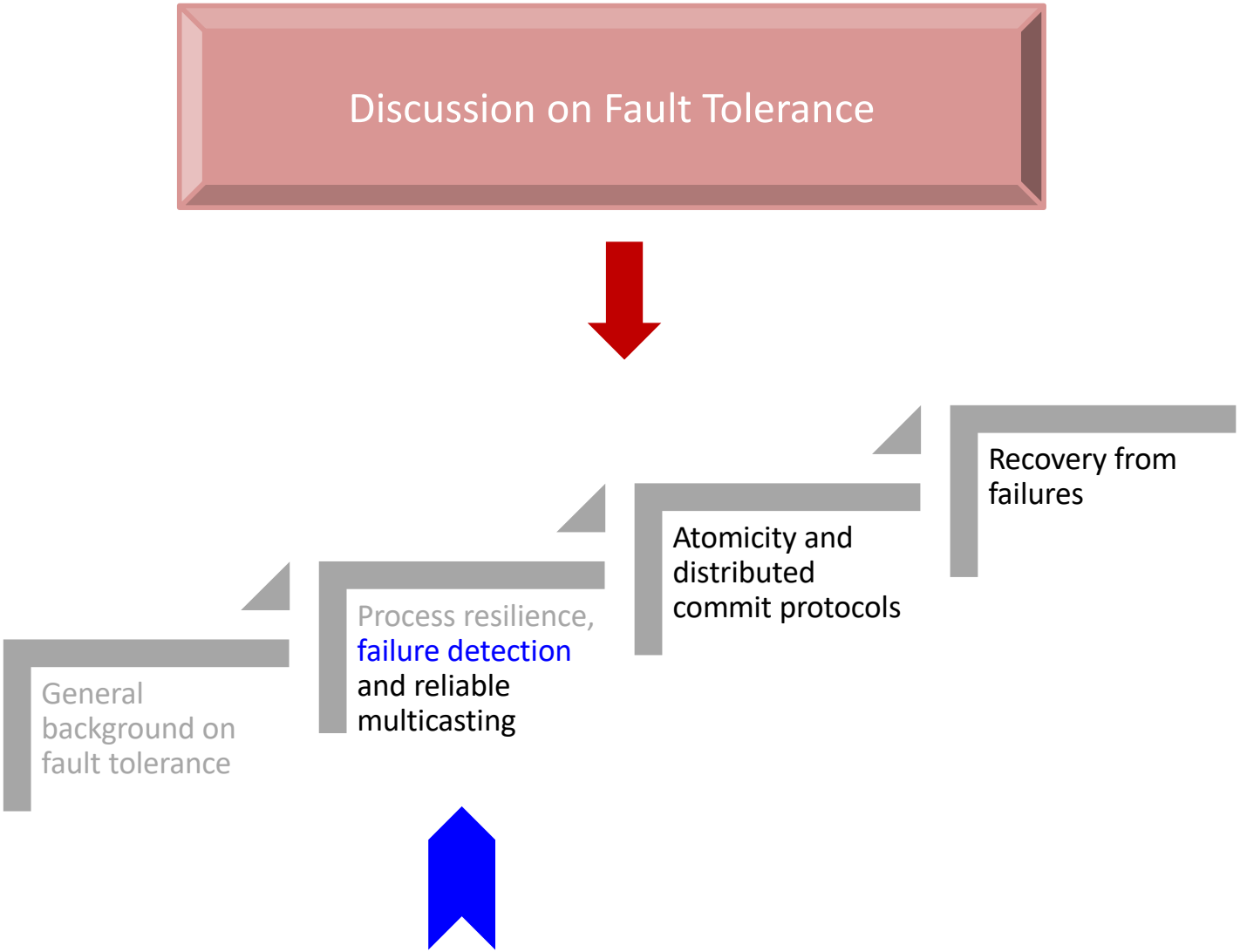
## Fault-Tolerance II

Olaf Landsiedel

# Last Time

- Fault Tolerance I
  - Failures
  - Process Resilience:
    - Byzantine Generals

# 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

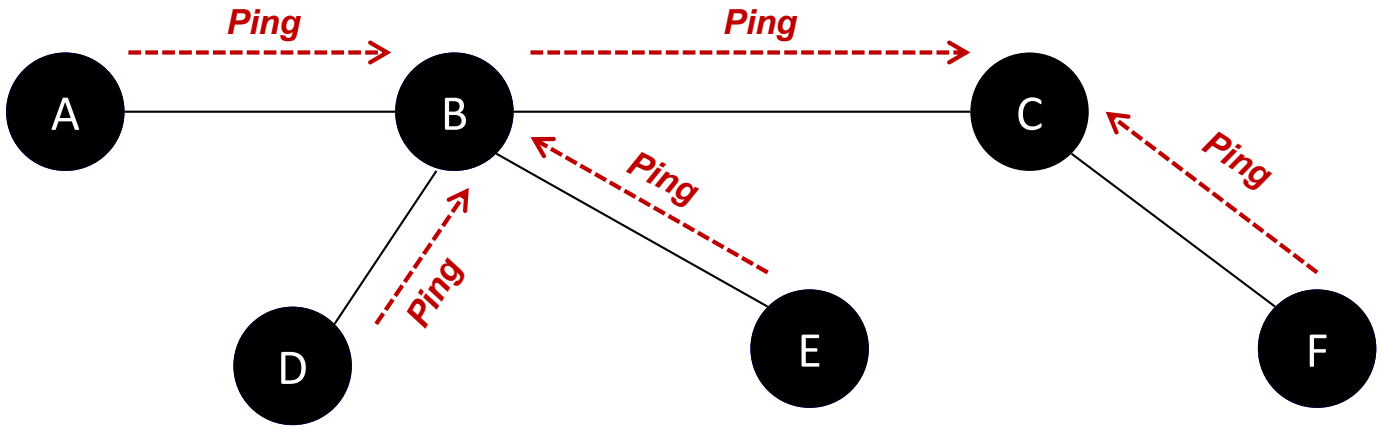
- Before we properly mask failures, we generally need to detect them
- For a group of processes, non-faulty members should be able to decide who is still a member and who is not
- Two policies:
  - Processes actively send "are you alive?" messages to each other (i.e., *pinging each other*)
  - Processes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

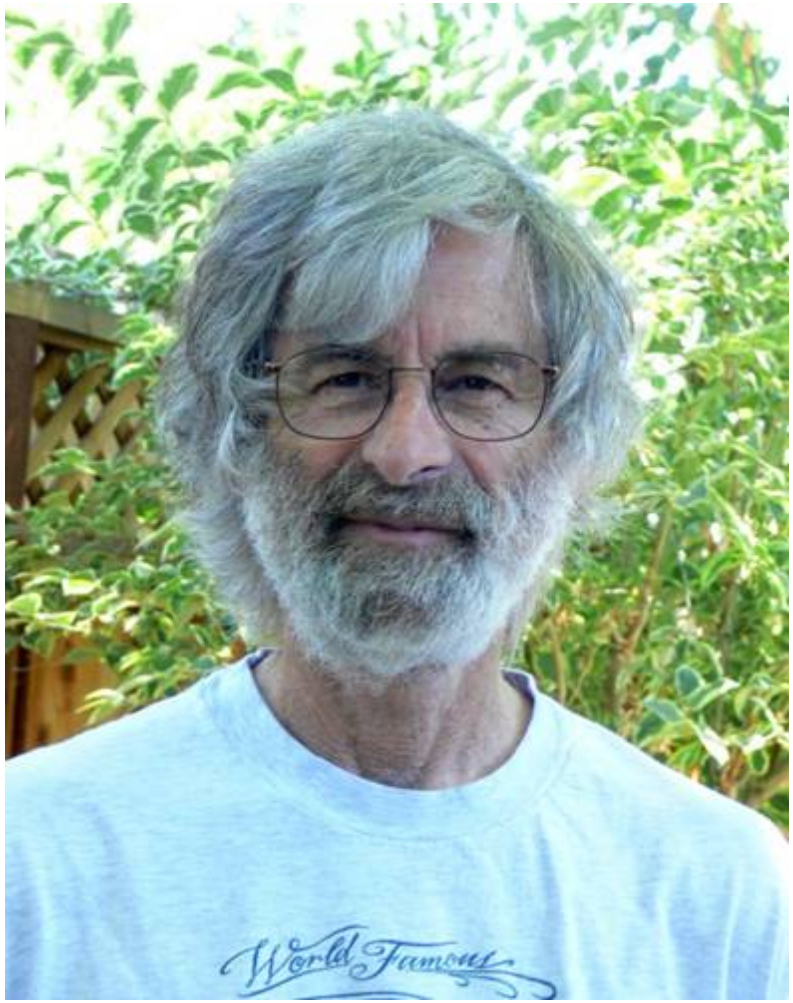
# Summary

- 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

# Leslie Lamport

*"A distributed system is one in which the failure of a machine you have never heard of can cause your own machine to become unusable"*

- Issue is dependency on critical components
- Notion is that state and "health" of system at site A is linked to state and health at site B

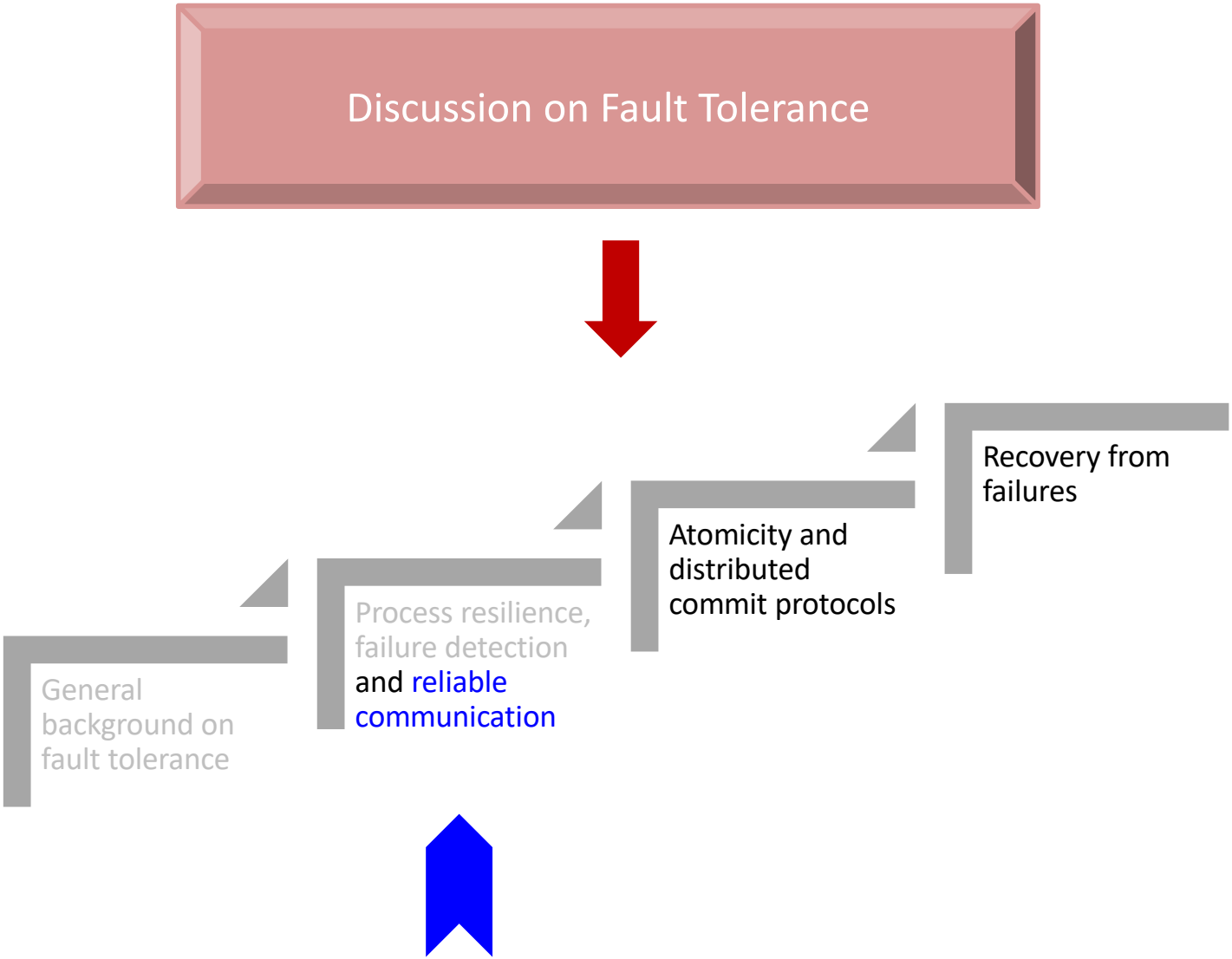



Picture: http://research.microsoft.com/en-us/um/people/lamport/

# Today…

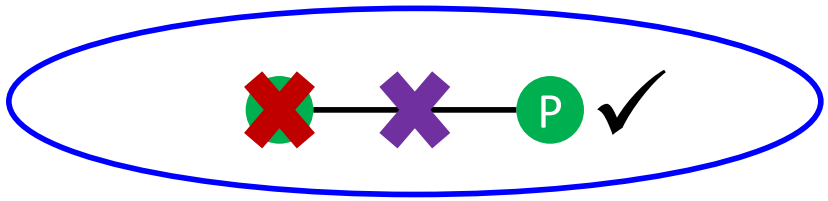
- Today's session
  - Fault Tolerance – *Part II*
    - Reliable communication

# Objectives


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

# Reliable Communication

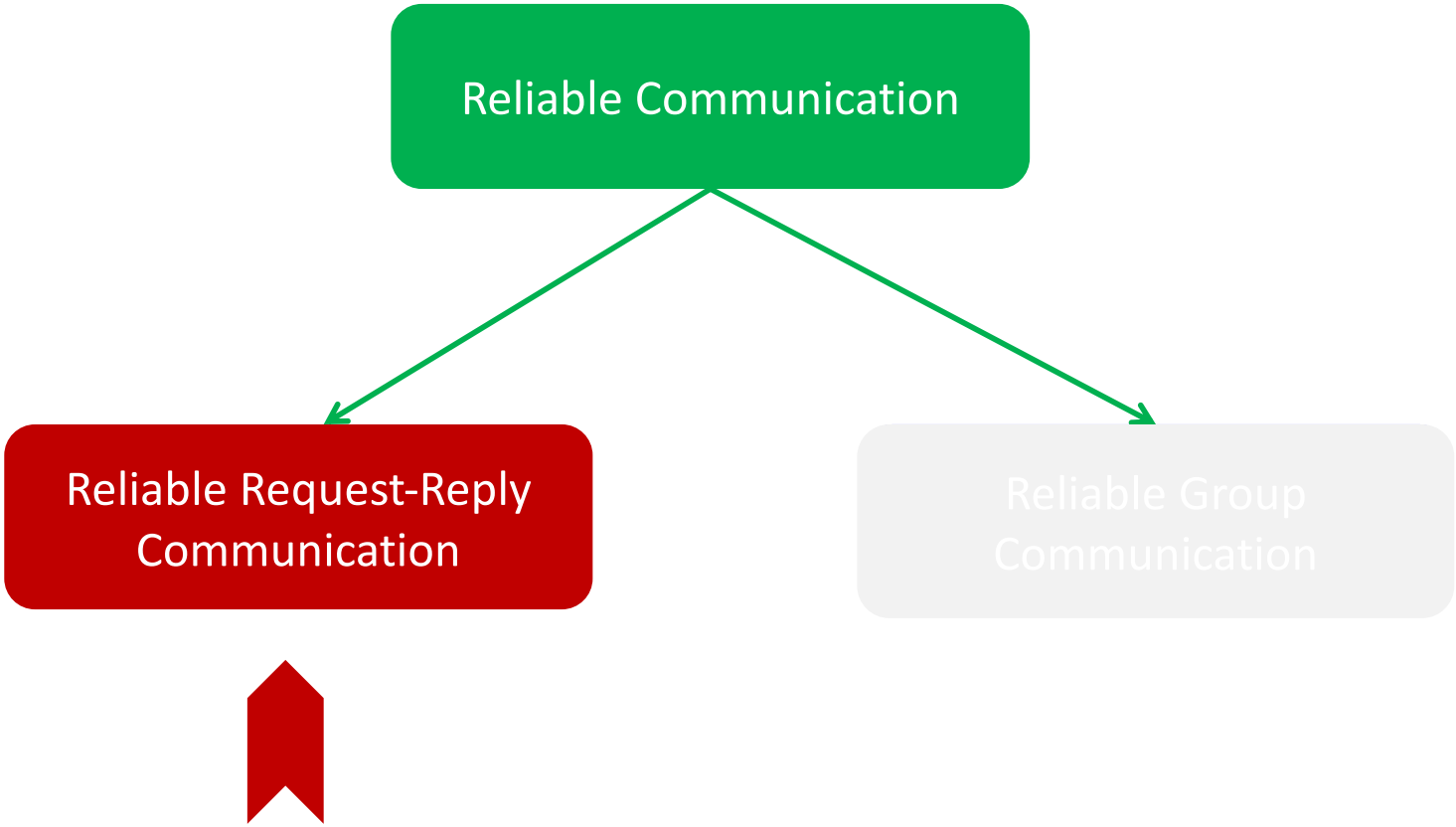
- Fault tolerance in distributed systems typically concentrates on faulty processes




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

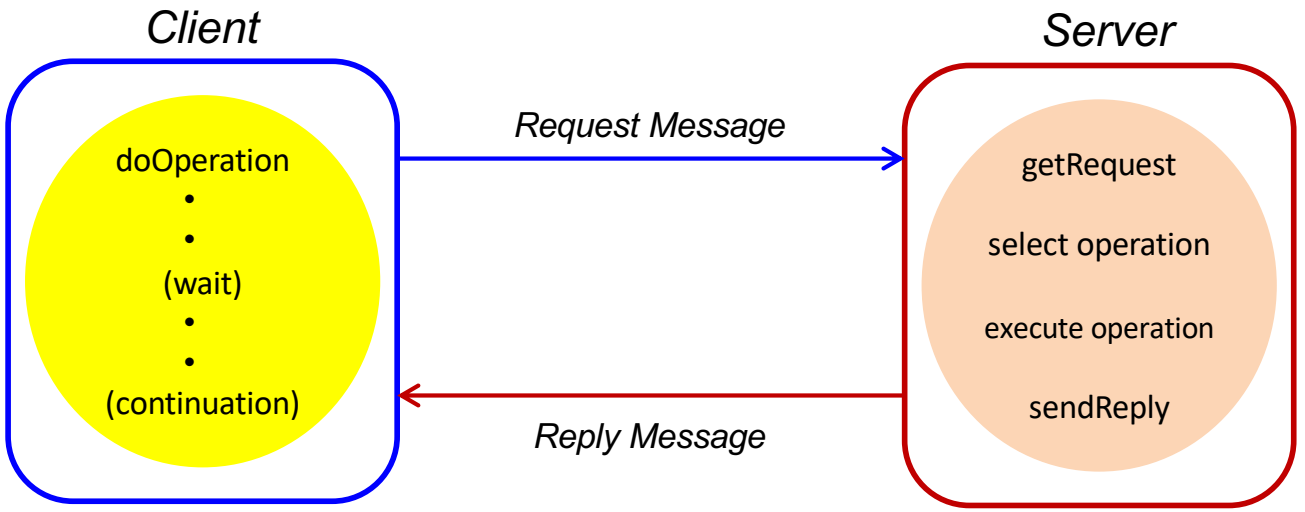
- We will focus on two types of reliable communication:
  - Reliable request-reply communication (e.g., RPC)
  - Reliable group communication (e.g., multicasting schemes)

# Reliable Communication



# 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:

# 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:

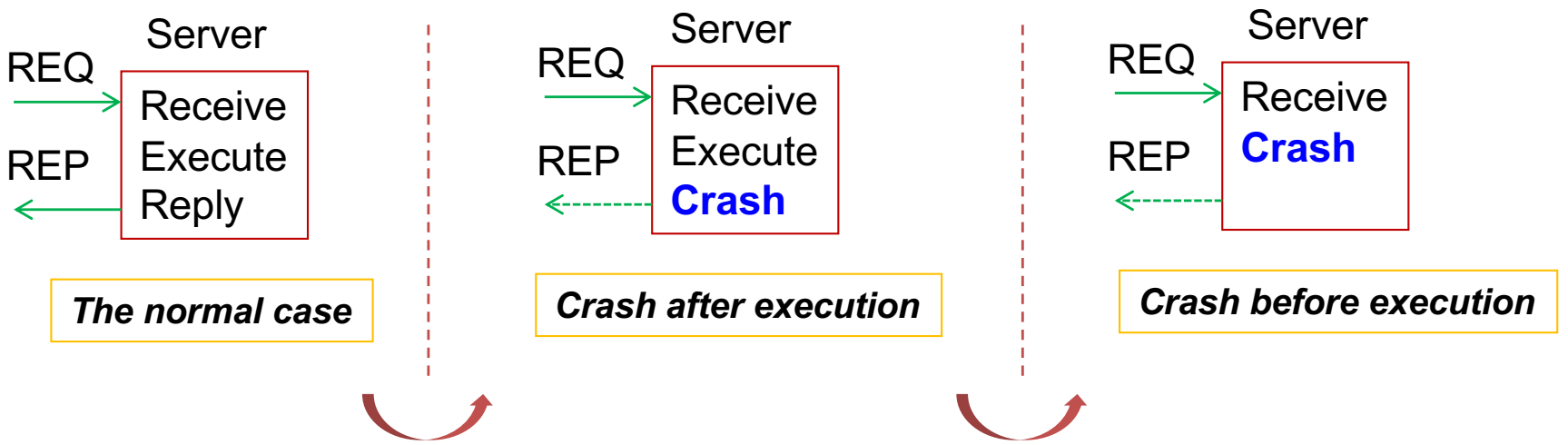
1. The client is unable to locate the server
2. The request message from the client to the server is lost
3. The server crashes after receiving a request
4. The reply message from the server to the client is lost
5. The client crashes after sending a request

# 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)*:

| Reissue Strategy | Strategy M→P | | | Strategy P→M | | |
|---|---|---|---|---|---|---|
| | **MPC** | **MC(P)** | **C(MP)** | **PMC** | **PC(M)** | **C(PM)** |
| **Always** | DUP | OK | OK | DUP | DUP | OK |
| **Never** | OK | ZERO | ZERO | OK | OK | ZERO |
| **Only when ACKed** | DUP | OK | ZERO | DUP | OK | ZERO |
| **Only when NOT ACKed** | OK | ZERO | OK | OK | DUP | OK |

# 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

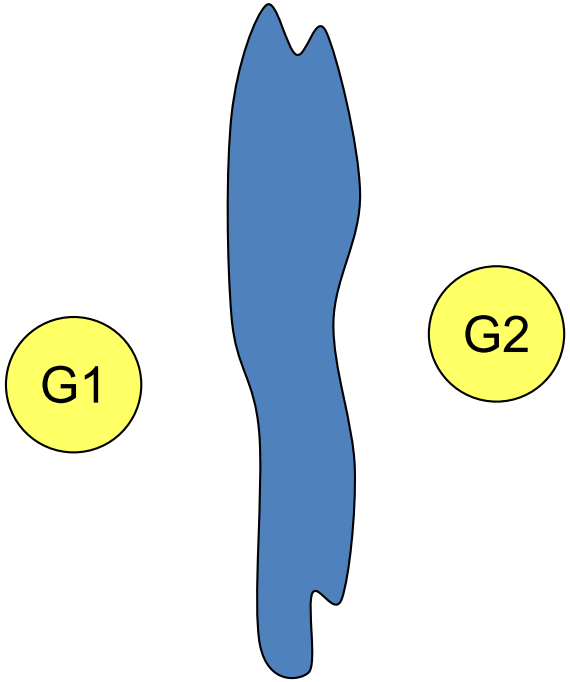
# Possible Solutions [Nelson 1981]

- ***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)***

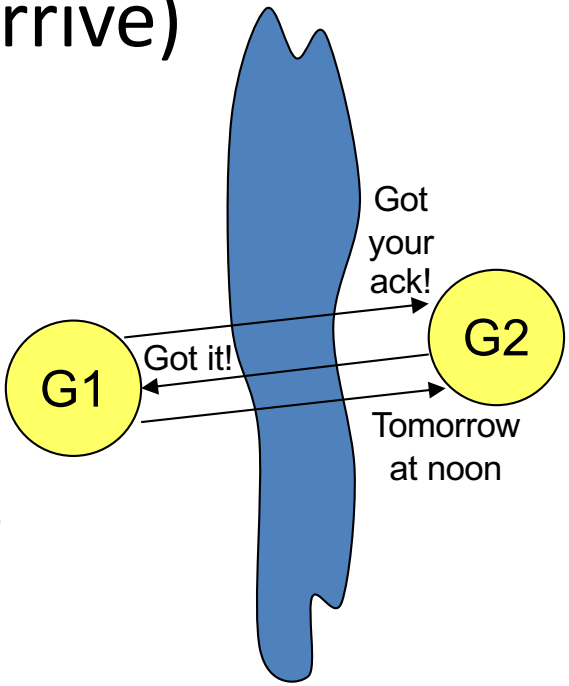
# The Two Generals Problem

- Two generals
  - lead divisions of an army
  - each camped on the mountains
  - on the two sides of an enemy-occupied valley
  - only communicate via messengers
- We need a scheme for the generals to agree on a common attack time
  - an attack by only one division would be disastrous
- Ideas?



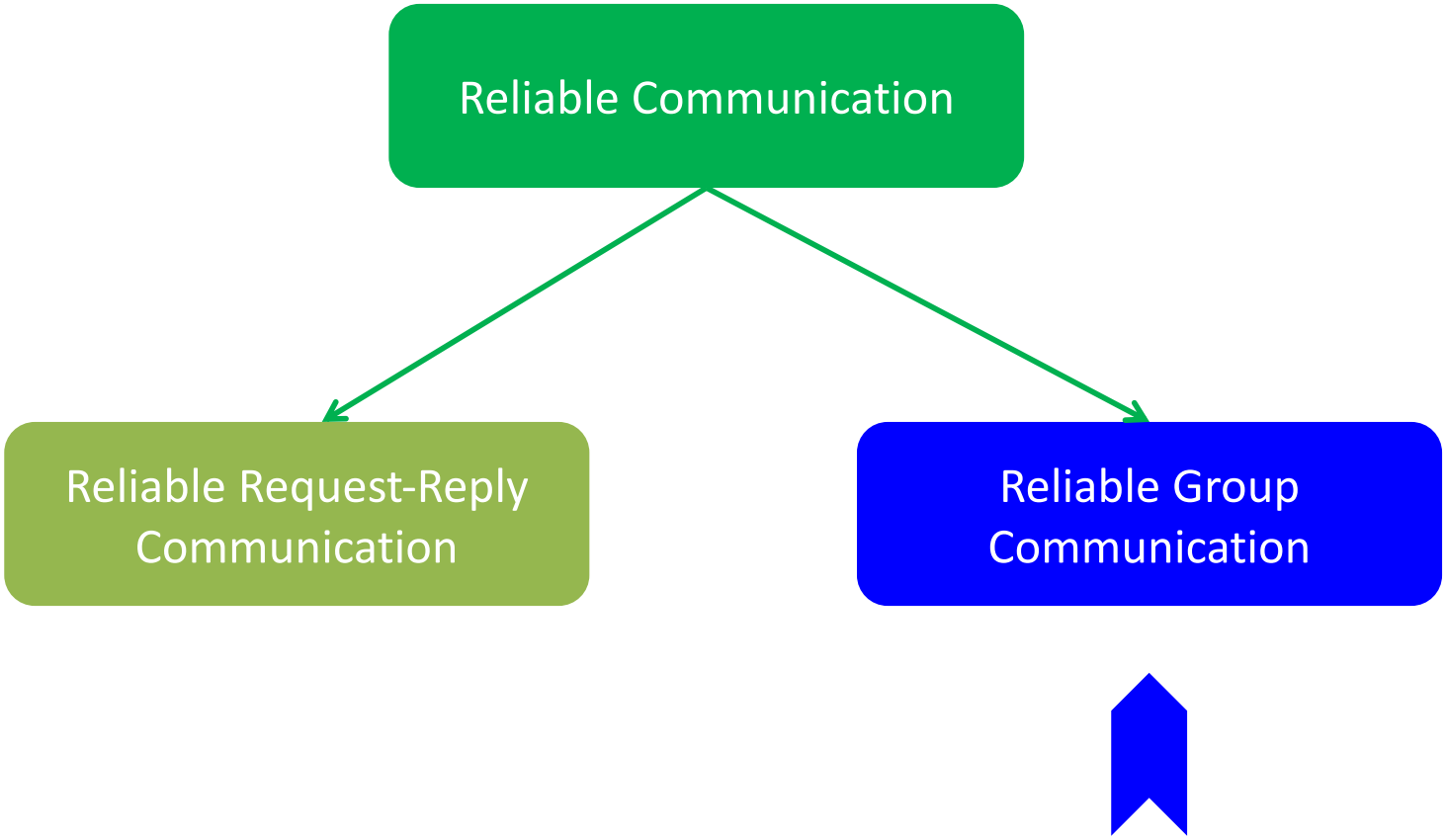
# The Two Generals Problem

- Messengers are totally reliable, but may need an arbitrary amount of time to cross the valley (they may even be captured and never arrive)
  - G1 decides on T,
    - sends a messenger to tell G2
  - G2 acknowledges receipt of the attack time T
  - G2, unsure whether G1 got the ack (without which he would not attack), will need an ack of the ack!



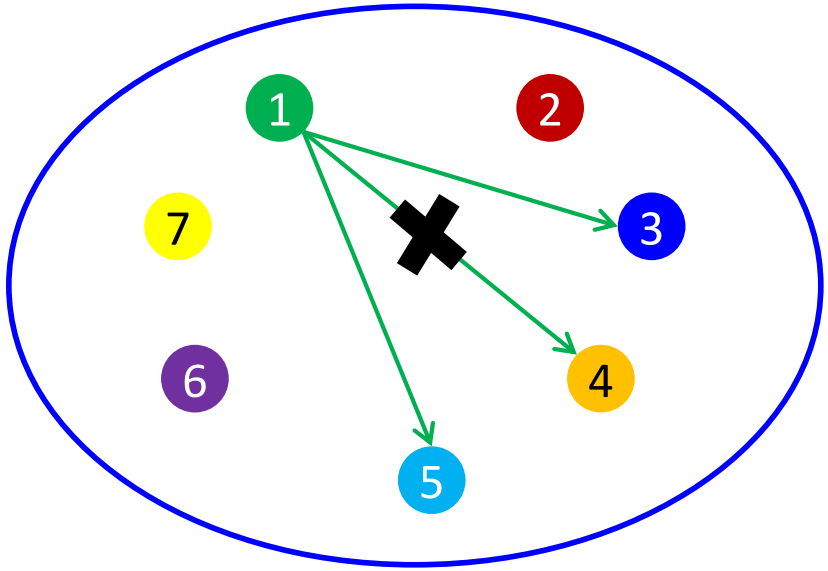

- This can go on forever, without either being sure

# Reliable 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

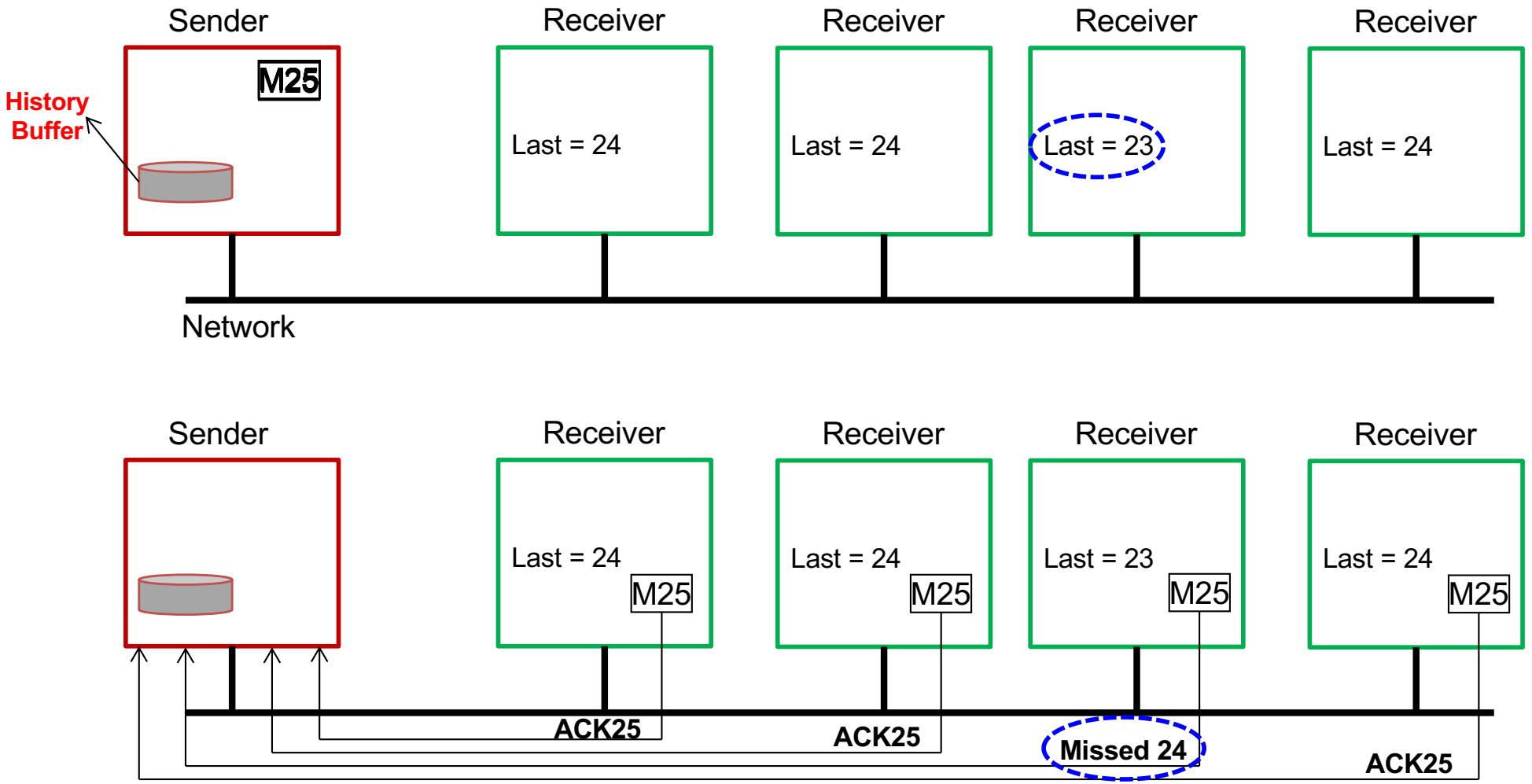
# Basic Reliable Multicasting Questions

- 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

- Example: Node S multicasts a message to multiple receivers
- Problem: message may be lost part way
  - delivered to some, but not to all, of the intended receivers
- Assumption: 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 Group Communication

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

# Scalability Issues with a Feedback-Based Scheme

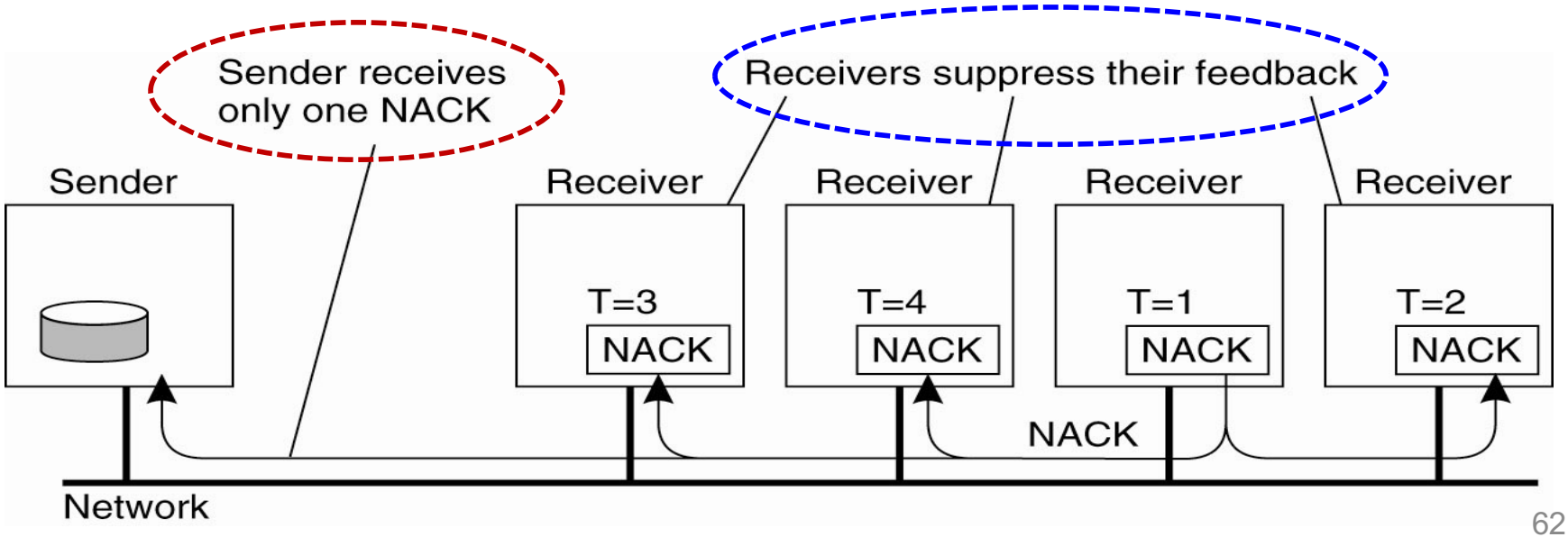
- If there are *N* receivers in a multicasting process, the sender must be prepared to accept at least *N* ACKs

- This might cause a feedback implosion
  - Instead, we can let a receiver return only a NACK

- 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

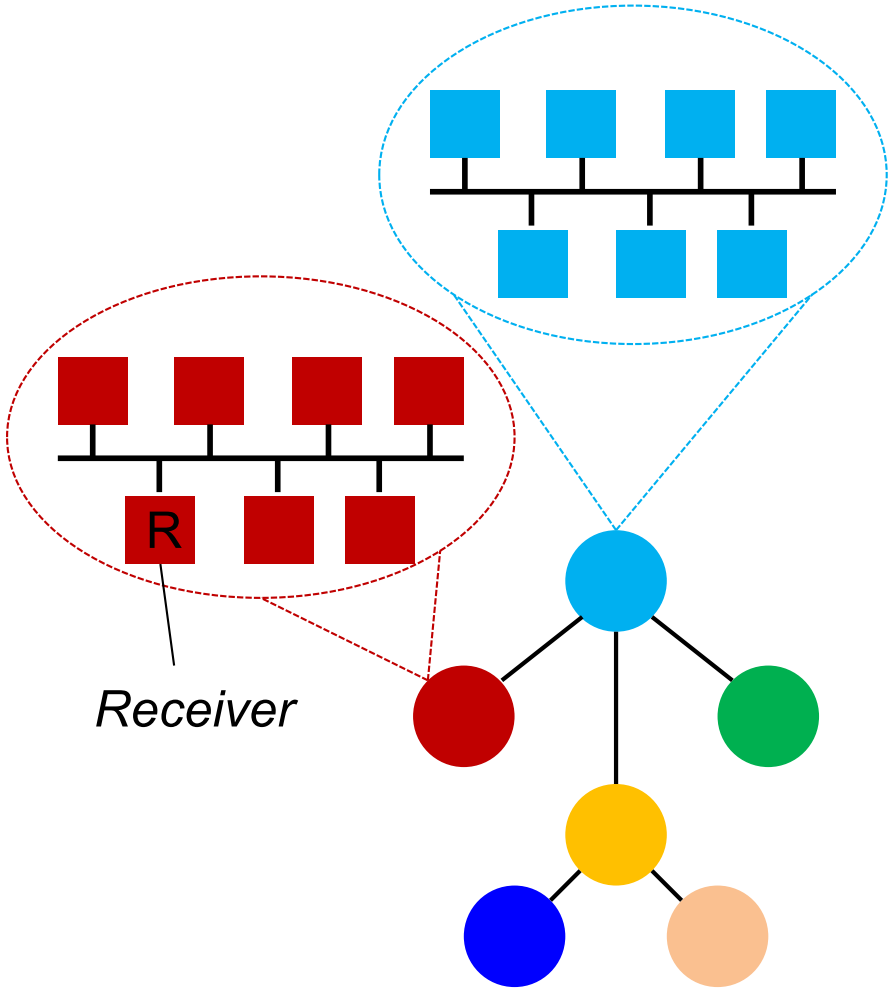
# Nonhierarchical Feedback Control

- 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



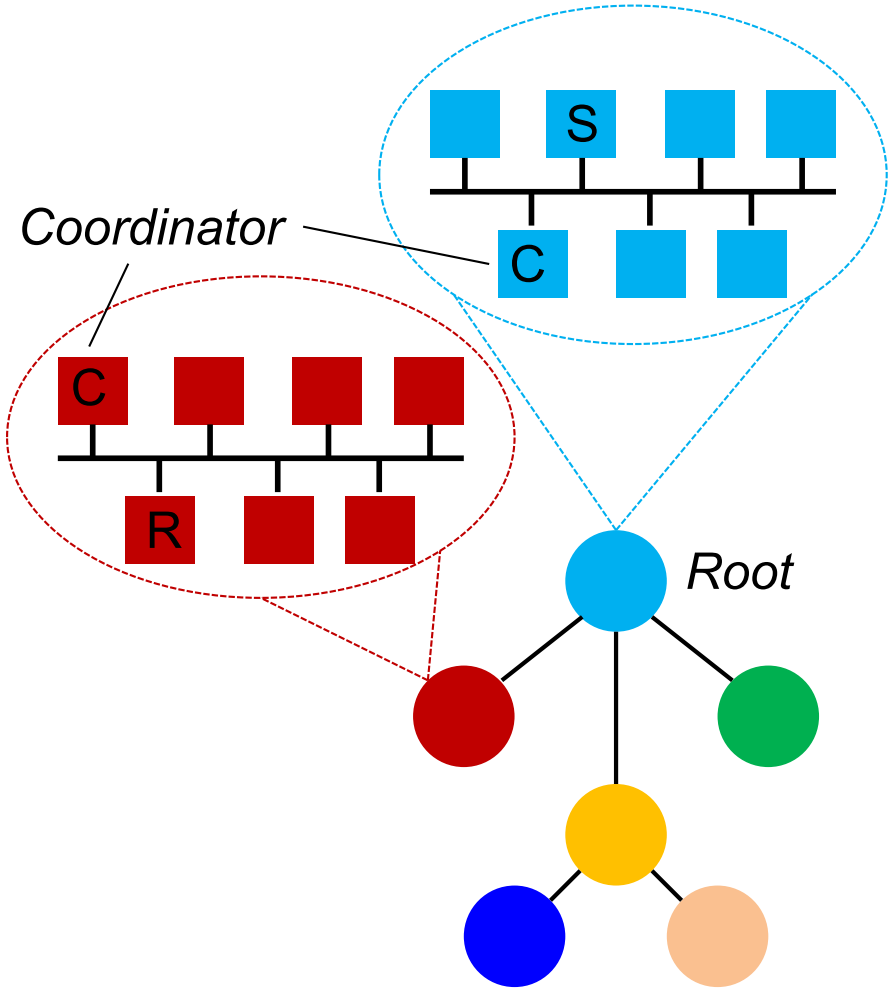
# Hierarchical Feedback Control

- Add structure (hierarchies)
  - Reduce feedback further
- Partition of receivers to subgroups
  - Organize as 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



# Next time:

- Continue with Reliable Group Communication
  - Atomic Multicast
- And much more
  - Atomicity and distributed commit protocols
  - Recovery from failures

# Questions?

In part, inspired from / based on slides from

- Mohammad Hammoud
- Muyuan Wang
- Philippas Tsigas