# Distributed Computing III

*Murphy was an optimist.*

*— O'Toole's Commentary*

*CSSE6400*

Richard Thomas

April 22, 2024

*Question*

# What communication faults may occur?

*Question*

# What communication faults may occur?

*Answer*

- Message not delivered

*Question*

# What communication faults may occur?

*Answer*

- Message not delivered
- Message delayed

*Question*

# What communication faults may occur?

*Answer*

- Message not delivered
- Message delayed
- Receiver failed

*Question*

# What communication faults may occur?

*Answer*

- Message not delivered
- Message delayed
- Receiver failed
- Receiver busy

*Question*

# What communication faults may occur?

*Answer*

- Message not delivered
- Message delayed
- Receiver failed
- Receiver busy
- Reply not received

*Question*

## What communication faults may occur?

*Answer*

- Message not delivered
- Message delayed
- Receiver failed
- Receiver busy
- Reply not received
- Reply delayed

*Question*

# How do we detect faults?

*Question*

# How do we detect faults?

*Answer*

- No listener on port – RST or FIN packet

*Question*

## How do we detect faults?

*Answer*

- No listener on port – RST or FIN packet
- Process crashes – Monitor report failure

*Question*

# How do we detect faults?

*Answer*

- No listener on port – RST or FIN packet
- Process crashes – Monitor report failure
- IP address not reachable – unreachable packet

*Question*

## How do we detect faults?

*Answer*

- No listener on port – RST or FIN packet
- Process crashes – Monitor report failure
- IP address not reachable – unreachable packet
- Query switches

*Question*

# How do we detect faults?

*Answer*

- No listener on port – RST or FIN packet
- Process crashes – Monitor report failure
- IP address not reachable – unreachable packet
- Query switches
- Timeout

*Question*

# What to do if fault is detected?

*Question*

What to do if fault is detected?

*Answer*

- Retry
- Restart

*Definition 1.* Idempotency

Repeating an operation does not change receiver's state.

# Byzantine Generals Problem

- $n$ generals need to agree on plan
- Can only communicate via messenger
- Messenger may be delayed or lost
- Some generals are traitors
  - Send dishonest messages
  - Pretend to have not received message
  - Send messages pretending to be another general

*Definition 2.* Byzantine Faults

Nodes in a distributed system may ‘lie’.

— Send faulty or corrupted messages or responses.

*Question*

Can we design a system to be Byzantine fault tolerant?

*Question*

Can we design a system to be Byzantine fault tolerant?

*Answer*

Yes, but, it is *challenging*.

# Limited Fault Tolerance

- Validate format of received messages
  - Need strategy to handle & report errors

# Limited Fault Tolerance

- Validate format of received messages
  - Need strategy to handle & report errors
- Santise inputs
  - Assume any input from external sources may be malicious

# Limited Fault Tolerance

- Validate format of received messages
  - Need strategy to handle & report errors
- Santise inputs
  - Assume any input from external sources may be malicious
- Retrieve data from multiple sources
  - If possible
  - e.g. Multiple NTP servers

*Assumption*

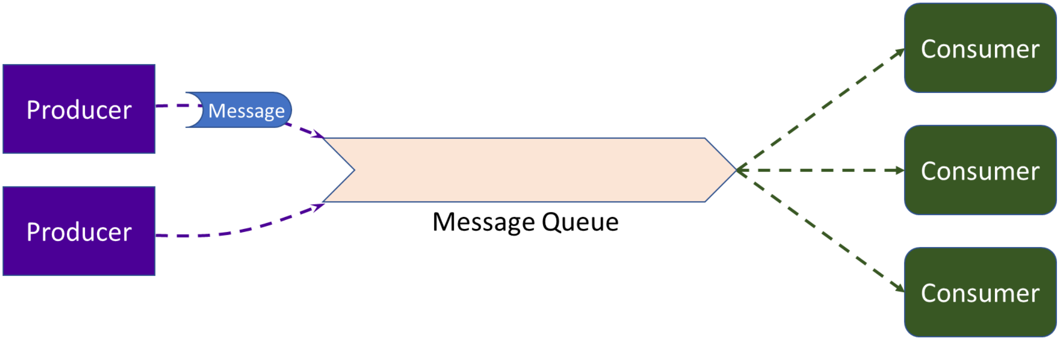
If all nodes are part of our system, we may assume there are no Byzantine faults.

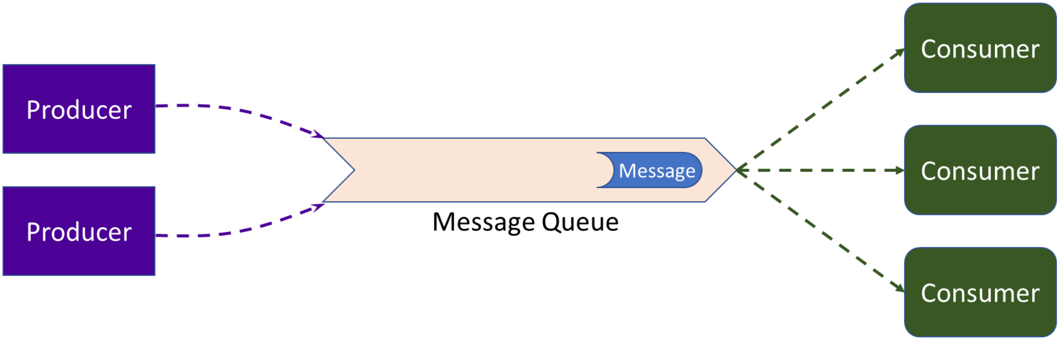
- Santise user input
- Byzantine faults may still arise
  - Logic defects
    - Same code is usually deployed to all replicated nodes, defeating easy fault tolerance solutions

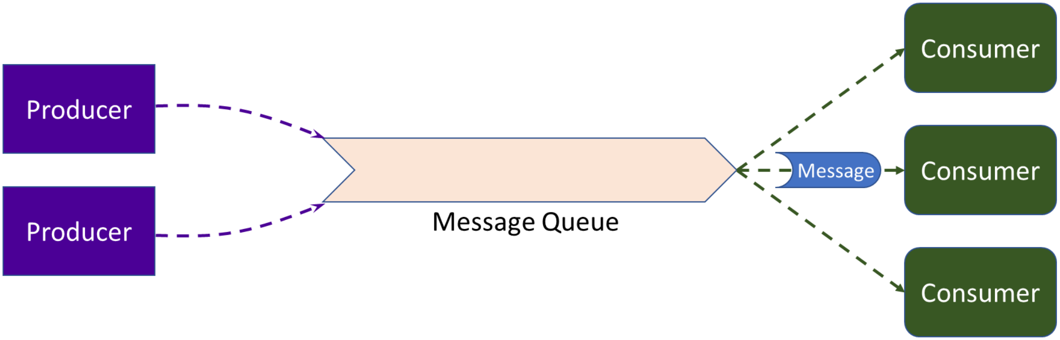
*Definition 3.* Poison Message

A message that causes the receiver to fail.

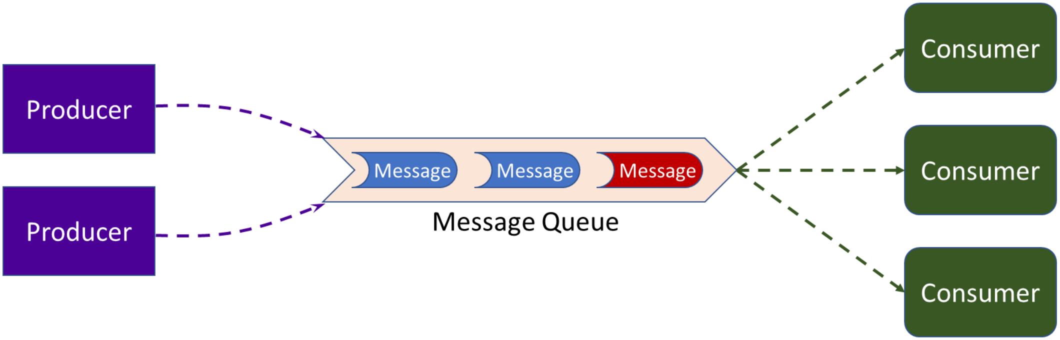
# Normal Message Flow

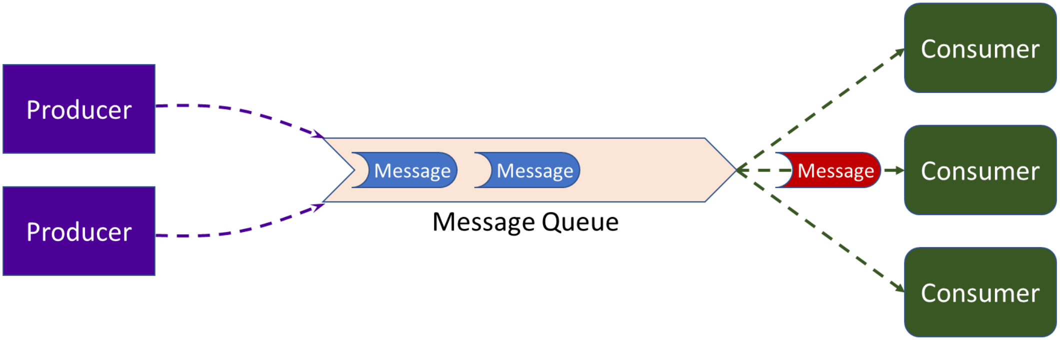

Producer
Message
Producer
Message Queue
Consumer
Consumer
Consumer

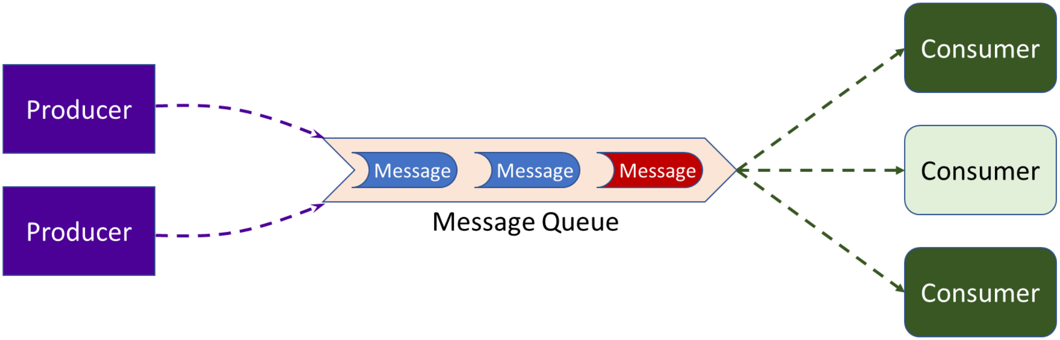

Producer
Producer
Message
Message Queue
Consumer
Consumer
Consumer

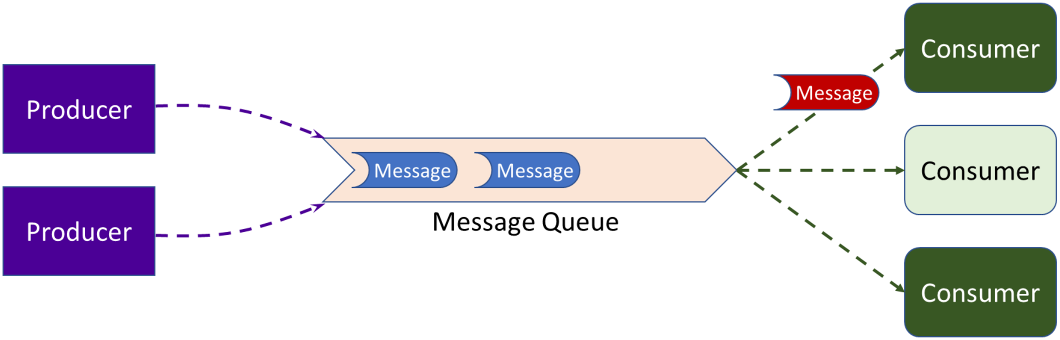

Producer
Producer
Message Queue
Message
Consumer
Consumer
Consumer

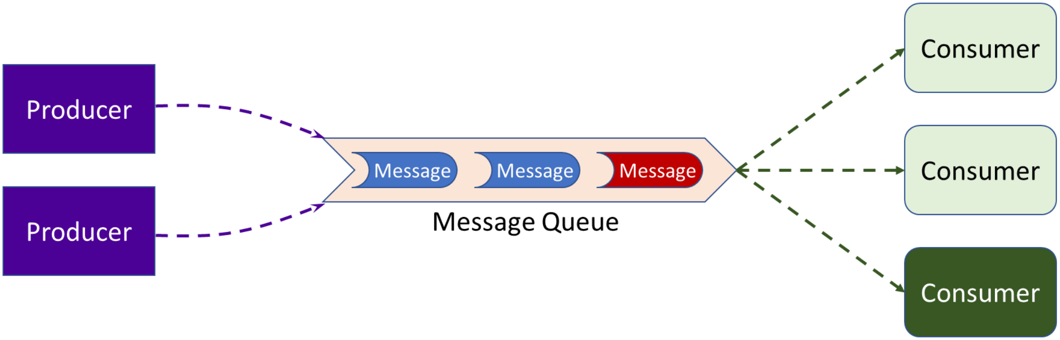
# Poison Message


Producer
Producer
Message
Message
Message
Message Queue
Consumer
Consumer
Consumer


Producer
Producer
Message
Message
Message Queue
Message
Consumer
Consumer
Consumer


Producer
Producer
Message
Message
Message
Message Queue
Consumer
Consumer
Consumer


Producer
Producer
Message
Message
Message Queue
Message
Consumer
Consumer
Consumer


Producer
Producer
Message
Message
Message
Message Queue
Consumer
Consumer
Consumer

*Question*

# What causes a message to be poisonous?

*Question*

# What causes a message to be poisonous?

*Answer*

- Content is invalid
  - e.g. Invalid product id sent to purchasing service
  - Error handling doesn't cater for error case

*Question*

# What causes a message to be poisonous?

*Answer*

- Content is invalid
  - e.g. Invalid product id sent to purchasing service
  - Error handling doesn't cater for error case
- System state is invalid
  - e.g. Add item to shopping cart that has been deleted
  - Logic doesn't handle out of order messages
    - Insidious asynchronous faults

# Detecting Poison Messages

Retry counter – with limit

- Where is counter stored?
  - Memory – What if server restarts?
  - DB – Slow
  - Must ensure counter is reset, regardless of how message is handled
    - e.g. Message is manually deleted

# Detecting Poison Messages

## Retry counter – with limit

- Where is counter stored?
  - Memory – What if server restarts?
  - DB – Slow
  - Must ensure counter is reset, regardless of how message is handled
    - e.g. Message is manually deleted

## Message service may have a timeout property

- Message removed from queue
  - Pending messages get older while waiting for poison message
  - Transient network faults may exceed timeout

# Detecting Poison Messages

## Monitoring service

- Trigger action if message stays at top of queue for too long
- Can check for queue errors
  - No messages are being processed
  - Restart message service

# Handling Poison Messages

## Discard message

- System must not require guarantee of message delivery
- Suitable when message processing speed is most important

# Handling Poison Messages

## Discard message

- System must not require guarantee of message delivery
- Suitable when message processing speed is most important

## Always retry

- Requires mechanism to fix message
    - Often requires manual intervention
- Suitable when message delivery is most important
- Very long delays in processing

# Handling Poison Messages

Dead-letter queue

- Long transient failures result in adding many messages
  - e.g. Network failure
- Requires manual monitoring and intervention
- System must not require strict ordering of messages
- Suitable when message processing speed is important

# Handling Poison Messages

## Retry queue

- Transient failures also added
- Use a previous strategy to deal with poison messages
- System must not require strict ordering of messages
- Suitable when message processing speed is very important
  - Main queue is never blocked
  - Receivers need to process from two message queues

*Definition 4.* Poison Pill Message

Special message used to notify receiver it should no longer wait for messages.

*Question*

# Why use a poison pill message?

*Question*

Why use a poison pill message?

*Answer*

Graceful shutdown of system.

*Question*

# How to order asynchronous messages?

*Question*

How to order asynchronous messages?

*Answer*

- Timestamps?
  - Can't keep clocks in sync
  - Limited clock precision

# § Data Issues

*Consistency*

Eventual Consistency weak guarantee

Linearisability strong guarantee

Causal Ordering strong guarantee

## *Eventual Consistency*

- Allows stale reads
- May be appropriate for some systems
  - e.g. Social media updates[1]

[1] See Distributed II slides 41 - 46.

*Linearisability*

- Once value is written, all reads see same value
  - Regardless of replica read from

- Once value is written, all reads see same value
  - Regardless of replica read from
- Single-leader replication
  - Read from leader
  - Read from synchronous follower

- Once value is written, all reads see same value
  - Regardless of replica read from
- Single-leader replication
  - Read from leader
  - Read from synchronous follower
- Multi-leader replication can't be linearised

## *Linearisability*

- Once value is written, all reads see same value
  - Regardless of replica read from
- Single-leader replication
  - Read from leader
  - Read from synchronous follower
- Multi-leader replication can't be linearised
- Leaderless replication
  - Lock value on quorum *before* writing

## *Causal Order*

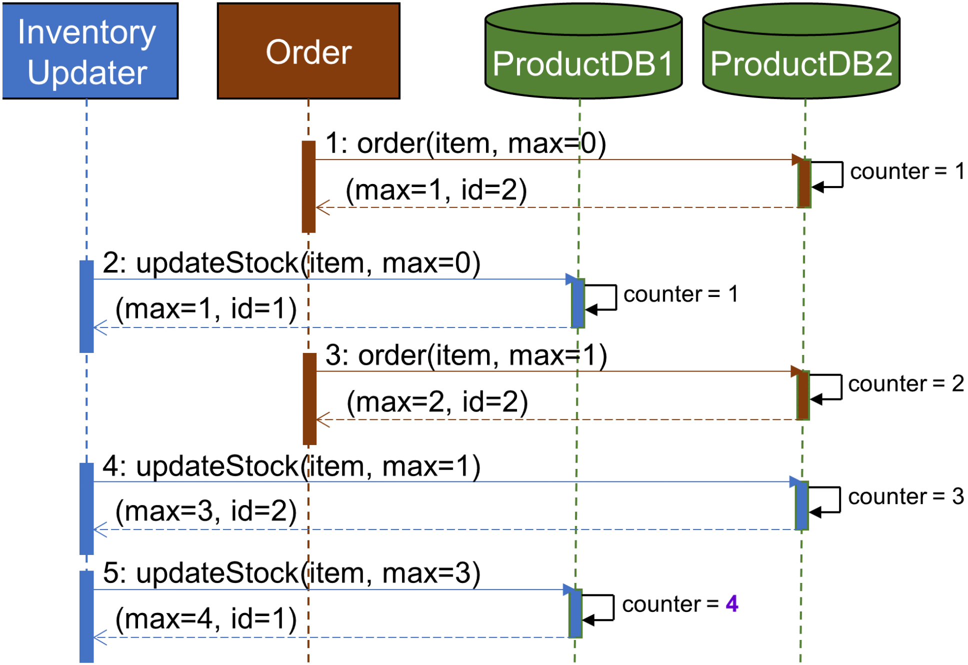
- Order is based on causality
  - What event needs to happen before another
  - Allows concurrent events

## *Causal Order*

- Order is based on causality
  - What event needs to happen before another
  - Allows concurrent events
- Single-leader replication
  - Record sequence number of writes in log
  - Followers read log to execute writes

## *Causal Order*

- Order is based on causality
  - What event needs to happen before another
  - Allows concurrent events
- Single-leader replication
  - Record sequence number of writes in log
  - Followers read log to execute writes
- Lamport timestamps


Inventory Updater
Order
ProductDB1
ProductDB2
1: order(item, max=0)
counter = 1
(max=1, id=2)
2: updateStock(item, max=0)
counter = 1
(max=1, id=1)
3: order(item, max=1)
counter = 2
(max=2, id=2)
4: updateStock(item, max=1)
counter = 3
(max=3, id=2)
5: updateStock(item, max=3)
counter = 4
(max=4, id=1)

*Definition 5.* Consensus

A set of nodes in the system agree on some aspect of the system's state.

## *Consensus Properties*

**Uniform Agreement** All nodes must agree on the decision

**Integrity** Nodes can only vote once

**Validity** Result must have been proposed by a node

**Termination** Every node that doesn't crash must decide

*Definition 6.* Atomic Commit

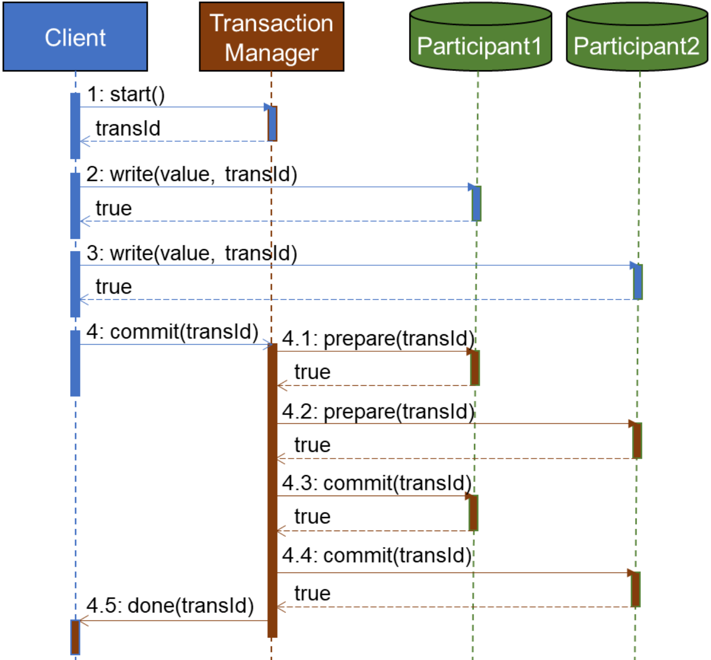
All nodes participating in a distributed transaction need to form consensus to complete the transaction.

*Two-Phase Commit*

**Prepare** Confirm nodes can commit transaction

**Commit** Finalise commit once consensus is reached

- Abort if consensus can't be reached



Client
Transaction Manager
Participant1
Participant2
1: start()
transId
2: write(value, transId)
true
3: write(value, transId)
true
4: commit(transId)
4.1: prepare(transId)
true
4.2: prepare(transId)
true
4.3: commit(transId)
true
4.4: commit(transId)
true
4.5: done(transId)

# Distributed Systems Timing Assumptions

- Synchronous System
  - Not realistic due to faults above
  - Minimal performance benefit

# Distributed Systems Timing Assumptions

- Synchronous System
  - Not realistic due to faults above
  - Minimal performance benefit
- Partially Synchronous System
  - Assumes important message order is preserved
  - Assumes most faults are rare & transient
  - Error handling to catch faults

# Distributed Systems Timing Assumptions

- Synchronous System
  - Not realistic due to faults above
  - Minimal performance benefit
- Partially Synchronous System
  - Assumes important message order is preserved
  - Assumes most faults are rare & transient
  - Error handling to catch faults
- Asynchronous System
  - No timing assumptions
  - Important message order managed by application
  - Difficult & limited design

# Distributed Systems Node Failure Assumptions

- Crash Stop
  - Node fails and never restarts

# Distributed Systems Node Failure Assumptions

- Crash Stop
  - Node fails and never restarts
- Crash Recovery
  - Node fails and restarts
    - Requires persistent memory for recovery close to prior state

# Distributed Systems Node Failure Assumptions

- Crash Stop
  - Node fails and never restarts
- Crash Recovery
  - Node fails and restarts
    - Requires persistent memory for recovery close to prior state
- Arbitrary Failure
  - Nodes may perform spurious or malicious actions
    - Byzantine faults

- Distributed systems are hard to build
- Large systems have to be distributed
  - Monoliths can't scale to millions of users
- Use environments, tools & libraries
  - Leaverage others' experience
- CSSE7610 Concurrency: Theory & Practice
  - Prove correctness of concurrent & distributed systems