

# Distributed Computing III

*Murphy was an optimist.*

— O'Toole's Commentary

CSSE6400

Richard Thomas

April 27, 2026

*Question*

What communication faults may occur?

*Question*

What communication faults may occur?

*Answer*

- Message not delivered

Lost in transit.

*Question*

What communication faults may occur?

*Answer*

- Message not delivered
- Message delayed

Network delay or receiver overloaded,  
but message will be processed later.

*Question*

What communication faults may occur?

*Answer*

- Message not delivered
- Message delayed  
Receiver software has crashed or node has died.
- Receiver failed

*Question*

What communication faults may occur?

*Answer*

- Message not delivered
- Message delayed  
Receiver temporarily not replying  
(e.g. garbage collection has frozen other processes).
- Receiver failed
- Receiver busy

*Question*

What communication faults may occur?

*Answer*

- Message not delivered
  - Message delayed
  - Receiver failed
  - Receiver busy
  - Reply not received
- Request was processed but reply lost in transit.

## *Question*

What communication faults may occur?

## *Answer*

- Message not delivered
  - Message delayed Reply will be received later.
  - 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

Assumes node is running & reachable.  
OS should close or refuse connection.  
Error packet may be lost in transit.

*Question*

How do we detect faults?

*Answer*

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

Assumes node is running & reachable.  
Most reliable.

*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

Router has to determine address is not reachable, which is no easier than for your application.

*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

Need permissions to do this.

Will only have this in your own data centre.

*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

UDP reduces network transmission time guarantee  
– does not perform retransmission.

*Question*

What to do if fault is detected?

*Question*

What to do if fault is detected?

*Answer*

- Retry
- Restart

- How many retries? How often?
- Exponential backoff with jitter
- How long to wait to restart?
- Too long reduces responsiveness.
- Unacknowledged messages need to be sent to other nodes – reducing performance.
- Too short may prematurely declare nodes dead.
- May lead to contention – two nodes processing the same request.
- May lead to cascading failure – load is sent to other nodes, slowing them down so they are then declared dead . . . .

### *Definition 0.* Idempotency

Repeating an operation does not change receiver's state.

- Idempotent consumer pattern
- Tag messages with an ID, so repeated messages can be ignored
- Or, redo messages that do not change state (e.g. queries)

## 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

Link analogy to Byzantine faults

### *Definition 0.* Byzantine Faults

Nodes in a distributed system may ‘lie’.

— Send faulty or corrupted messages or responses.

- A message that causes the receiver to fail.
- Incorrect responses (e.g. they have finished processing a message but haven’t).
- Can be due to faults or malicious hosts.
- Difficult to deal with all possible variations of these faults.

*Question*

Can we design a system to be Byzantine fault tolerant?

*Question*

Can we design a system to be Byzantine fault tolerant?

- Most systems don't attempt to
- Some need to (e.g. safety critical systems, blockchain, ...)
- Refer to CSSE3012 Safety Critical guest lecture.

*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
- Sanitise inputs
  - Assume any input from external sources may be malicious

## Limited Fault Tolerance

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

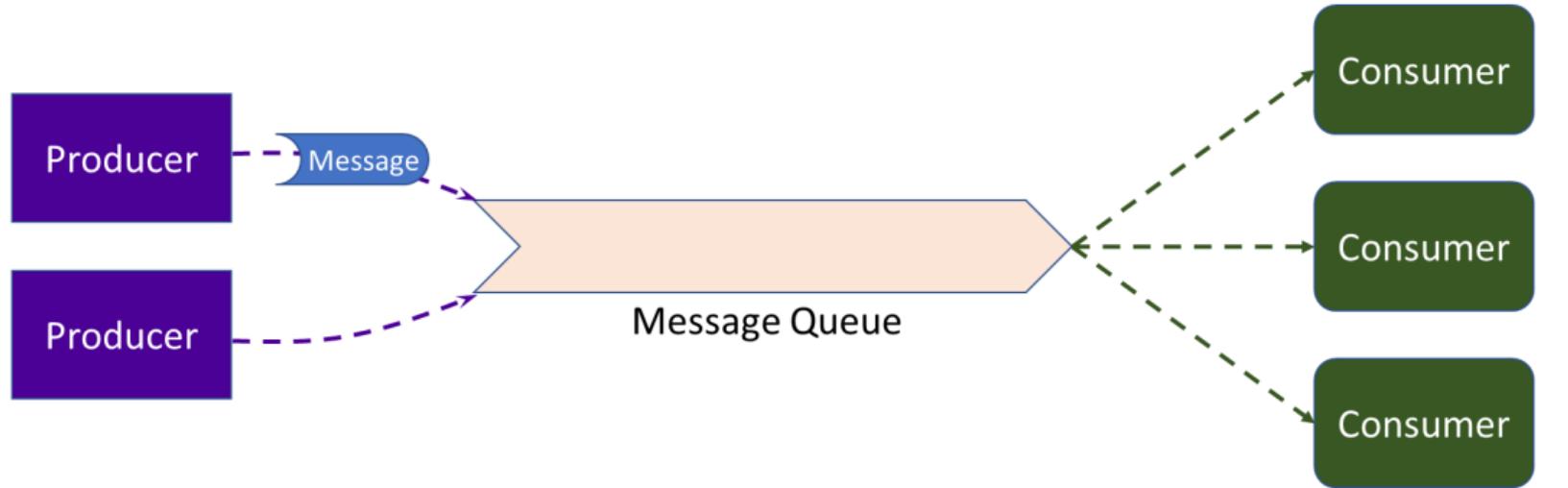
- Sanitise user input
- Byzantine faults may still arise
  - Logic defects
    - Same code is usually deployed to all replicated nodes, defeating easy fault tolerance solutions

### *Definition 0.* Poison Message

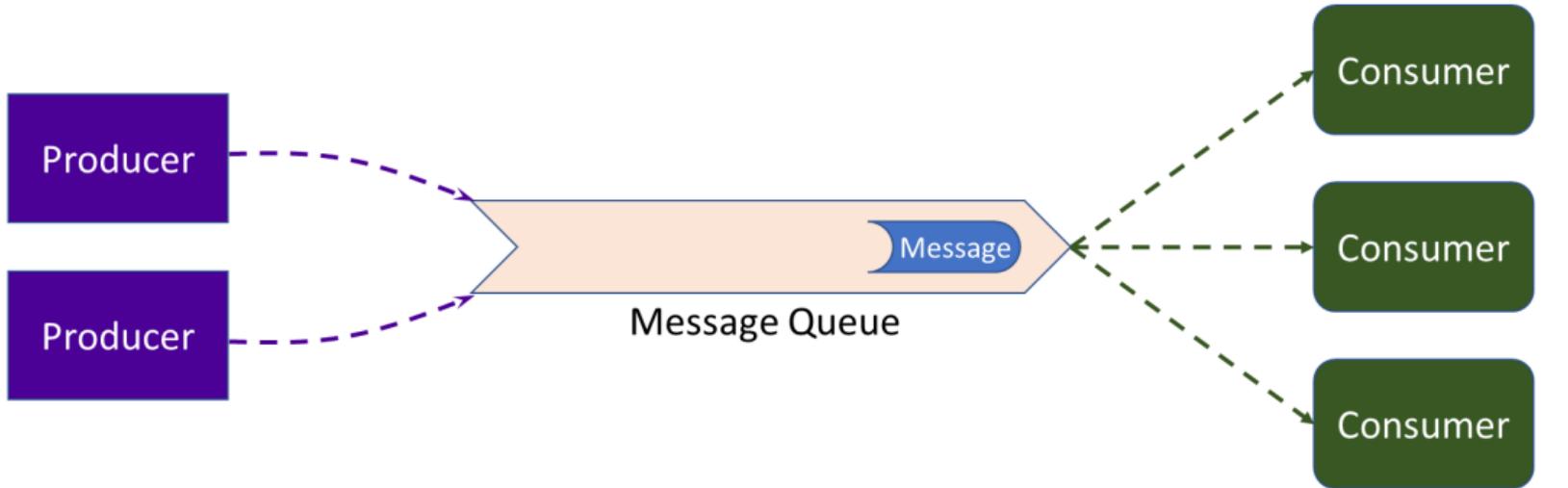
A message that causes the receiver to fail.

- Could literally cause the receiver to crash
- Often the receiver just cannot process the message and aborts processing

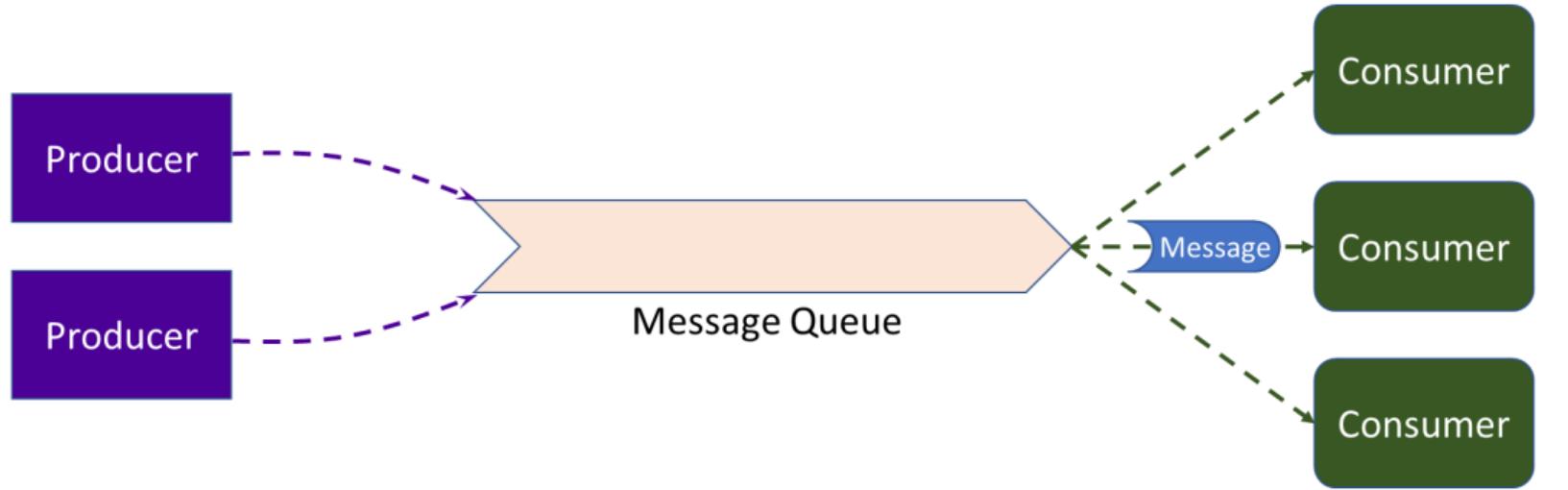
## Normal Message Flow



- Normal message/event is sent from a Producer.



- Normal message/event is queued (in Message Queue).

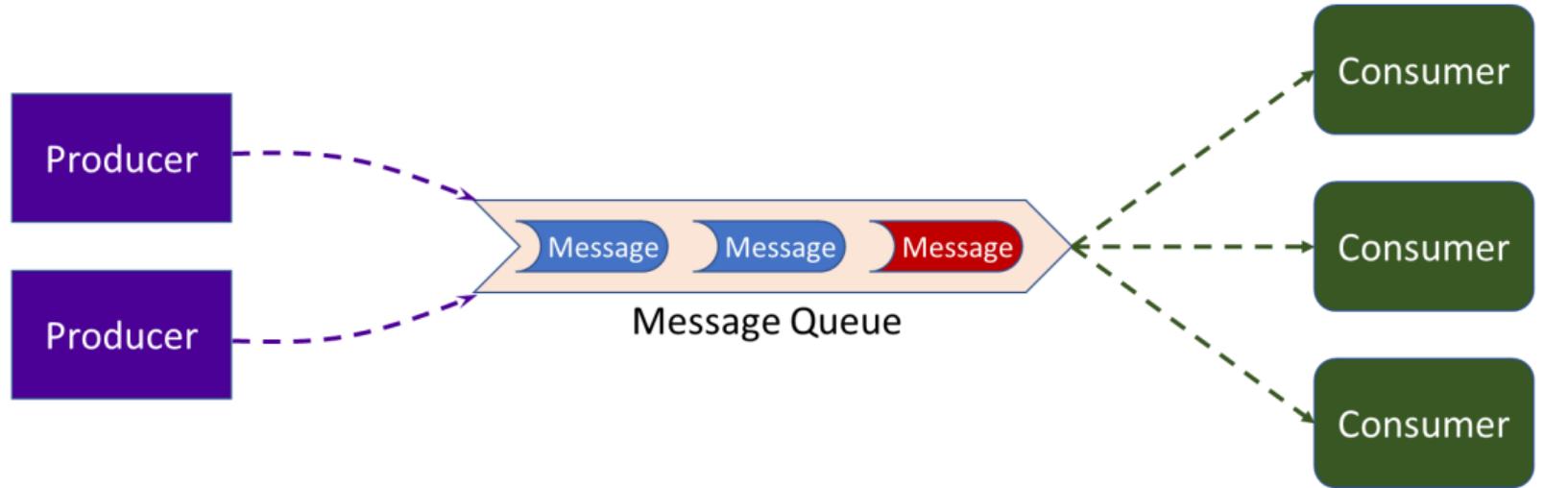


- Normal message/event is dequeued and processed by a Consumer.

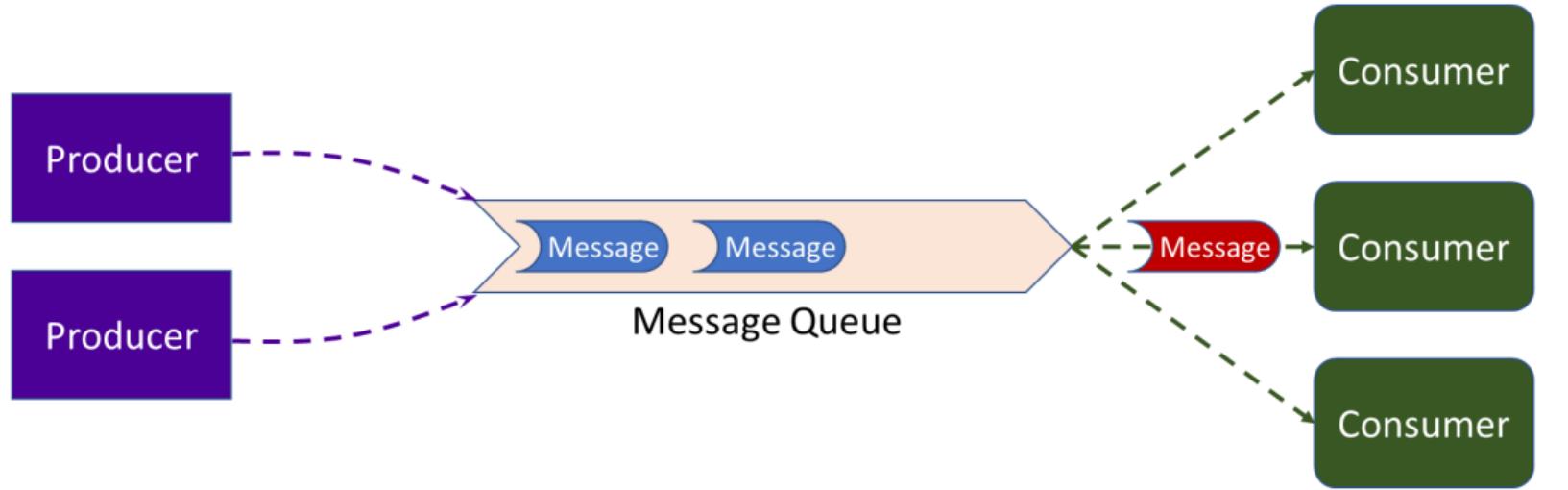
## Poison Message



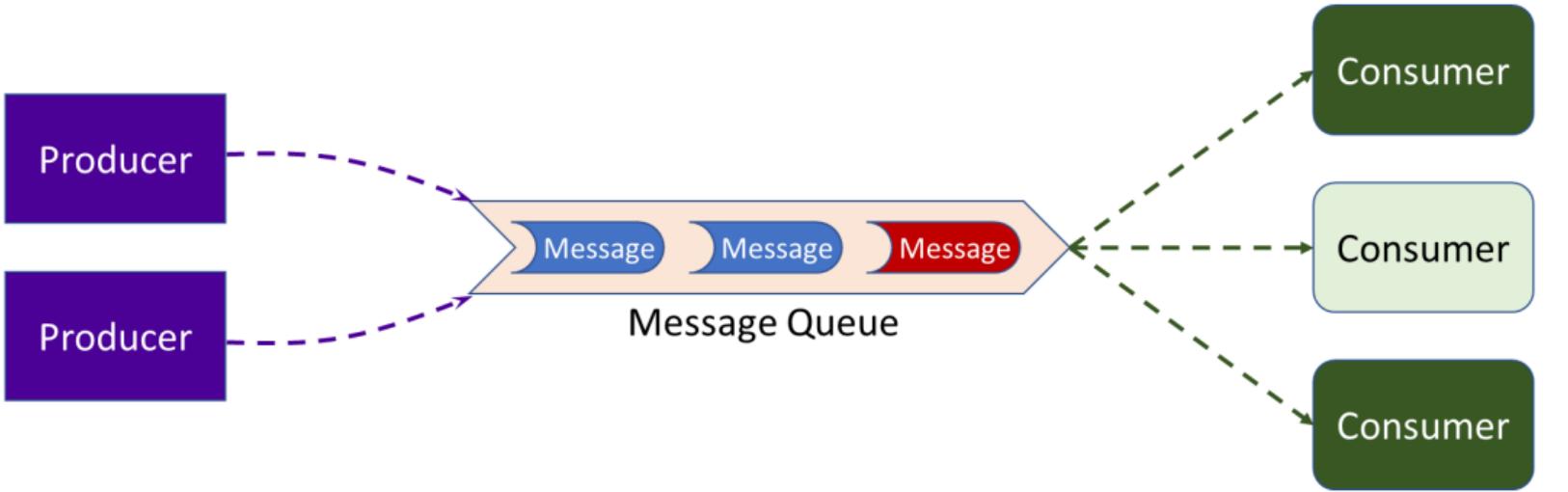
- Receiver can't process message.
- Always fails – Not due to transient failure.
- Failed messages are retried.
- Returned to front of queue – Preserve message order.
- Next receiver fails to process message – Infinite loop.
- Blocks sending of following messages.



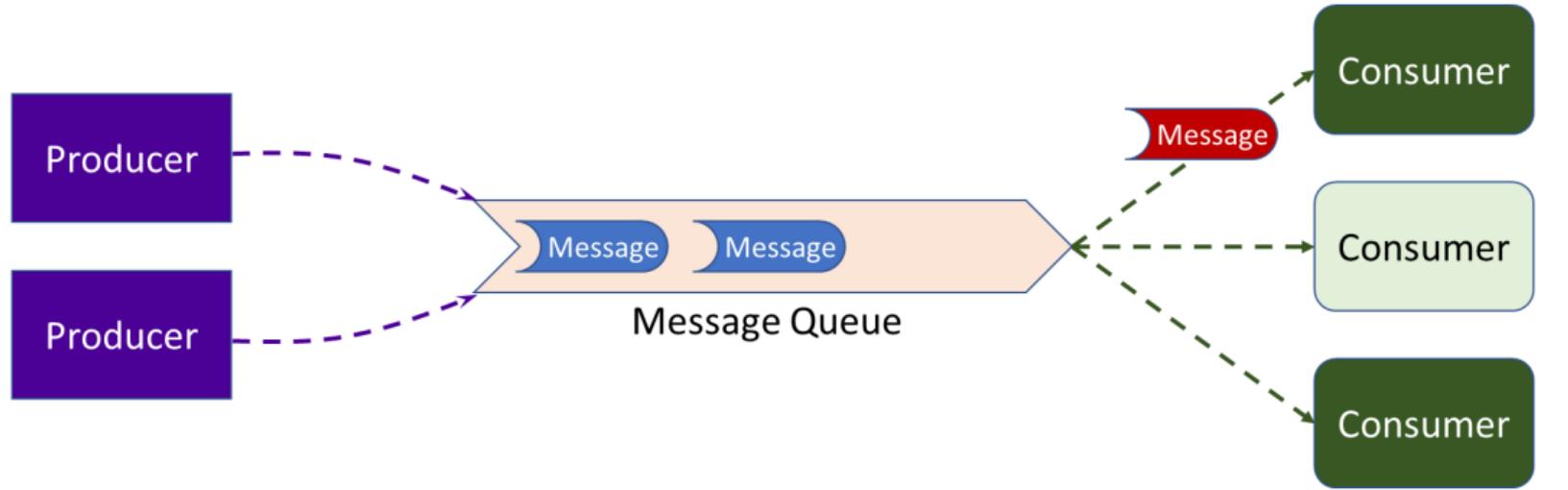
- This set of slides is an example of a poison message blocking the queue.
- Poison message is at head of queue. blocking issue.



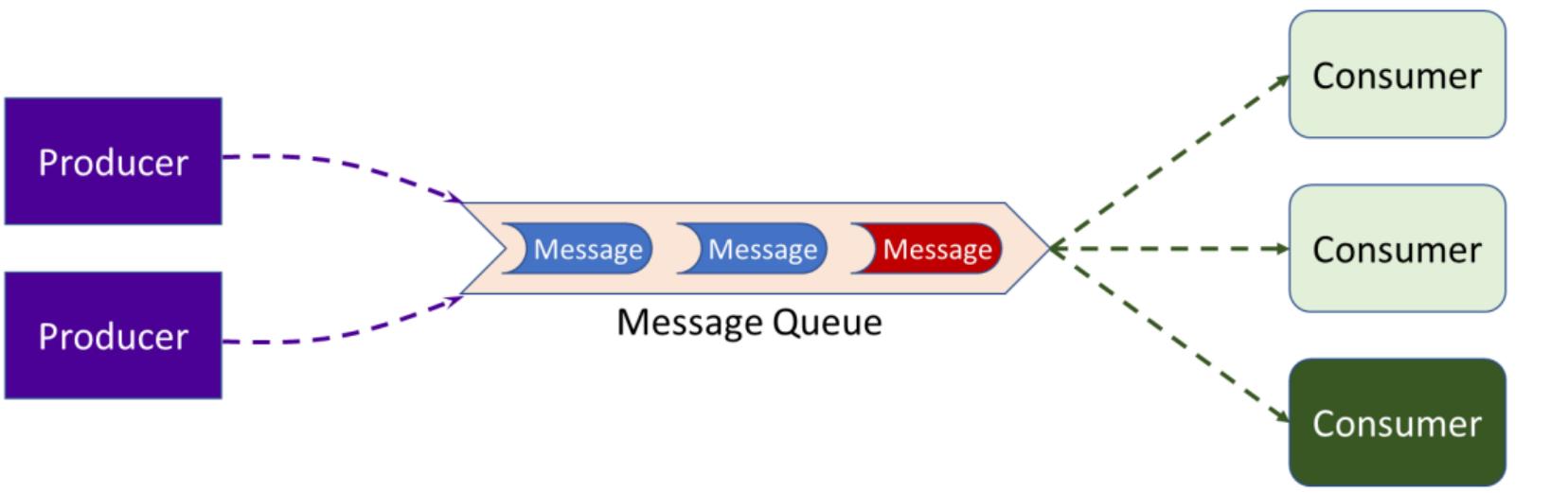
- This set of slides is an example of a poison message blocking the queue.
- Poison message is dequeued by a Consumer.



- This set of slides is an example of a poison message blocking the queue.
- Consumer fails (crashes).
- Poison message is added back to the head of the queue (re-try).



- This set of slides is an example of a poison message blocking the queue.
- Next Consumer dequeues poison message and fails (crashes).



- This set of slides is an example of a poison message blocking the queue.
- Poison message is added back to the head of the queue again (re-try).
- Infinite loop ...
- **Comment** that poison messages block processing regardless of how they're delivered.
- A message queue or service isn't the key blocking point.
- Async messages sent directly to a consumer requires it to queue them as they're processed, leading to the same blocking issue.

*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

Invalid content may be

- corrupted data,
- old version of data structure,
- incorrect data, or
- malicious data.

*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

Invalid state may be

- events out of order (e.g. delete then update),
- logic error making state invalid, or
- external corruption of persistent state.

## 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 0.* Poison Pill Message

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

Emphasise that this is **different** to a poison message

*Question*

Why use a poison pill message?

*Question*

Why use a poison pill message?

*Answer*

Graceful shutdown of system.

- Implementation is challenging with multiple producers and/or consumers.
- It must be the last message received by *all* consumers.

*Question*

How to order asynchronous messages?

*Question*

How to order asynchronous messages?

*Answer*

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

- Trying to sync with NTP is unreliable
- Network delays during sync
- Clock drift between syncs
- Finite precision – two events may end up with the same timestamp, if they occur in quick succession

## *§ 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<sup>1</sup>

---

<sup>1</sup>See Distributed II slides 45 - 50.

## *Linearisability*

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

Abstraction over replicated database.

Used when uniqueness needs to be guaranteed.

## *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
- SLR – defeats most performance benefits.  
Maintains reliability benefits.

## *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

## *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

Leaderless – similar performance cost to SLR.

## *Causal Order*

- Order is based on causality
  - What event needs to happen before another
  - Allows concurrent events
- Linearisation defines a *total* order.
- Causal ordering defines a *partial* order.
- e.g. Git repo history with branching as *causal order*.
- Not as strict as linearisability, so less performance cost.

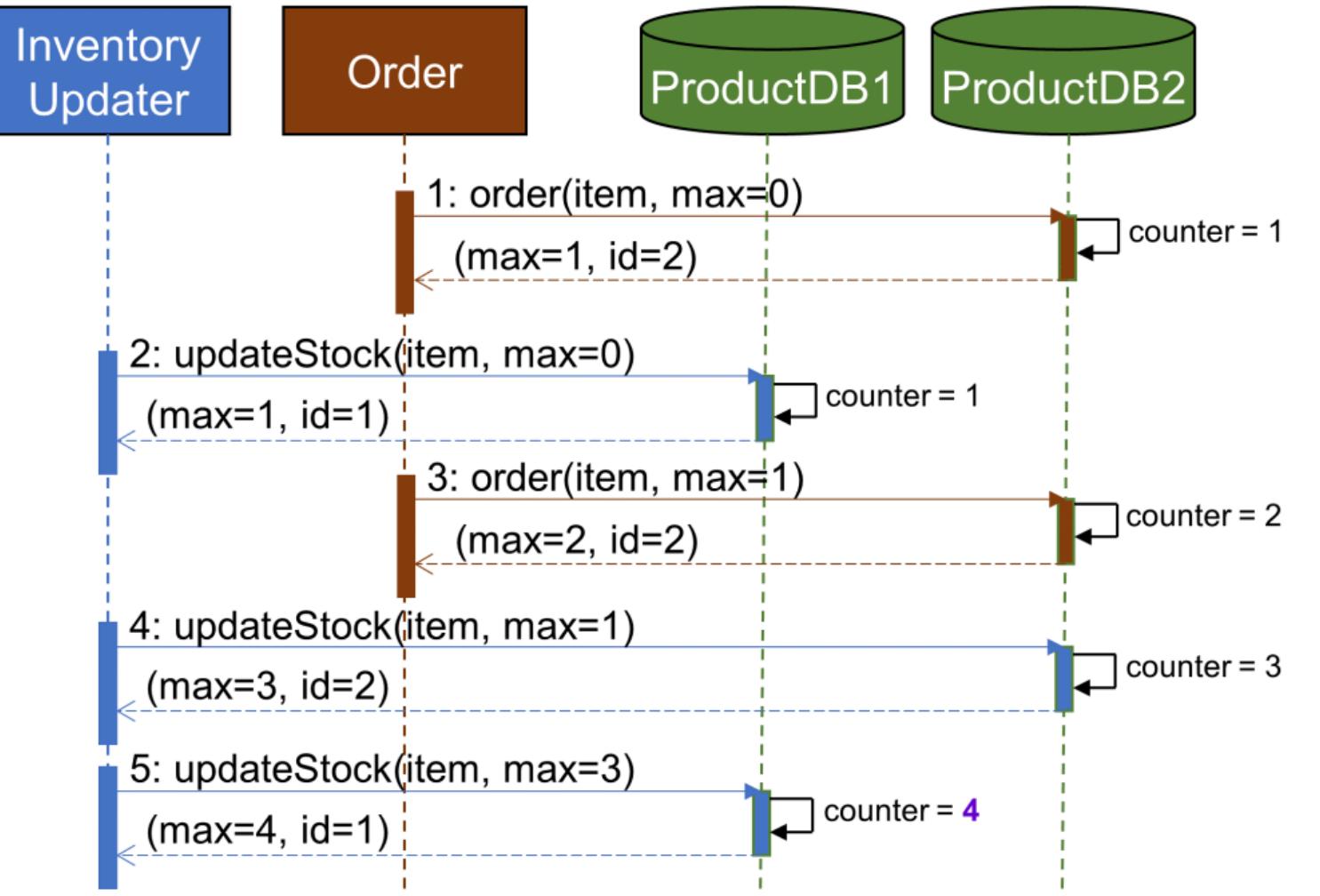
### *Causal Order*

- 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

Next slide is an example of Lamport timestamps.



- Each node has an id, & counts number of operations.
- Timestamp is a tuple (*counterValue, nodeID*).
- *nodeID* guarantees every timestamp is unique, even if they have the same counter value.
- Every node stores the maximum counter value seen so far.
- Maximum is passed in every request to another node.
- If a node receives a request or response with a maximum counter value greater than its own counter value, it increases its own counter to the new maximum.
- **Message 4:** InventoryUpdater sends its current maximum value (1). ProductDB2's counter is 2, so it increases it to 3 & returns to InventoryUpdater. InventoryUpdater records 3 as its new maximum value. **Message 5:** Updates ProductDB1's max value & then counts operation.

### *Definition 0.* Consensus

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

- Abstraction to make it easier to reason about system state.
- e.g. Selecting Leader DB if leader fails, or some Followers think it has failed.

## *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

- *Uniform Agreement* and *Integrity* are key
- *Validity* avoids nonsensical solutions
  - e.g. Always agreeing to a null decision
- *Termination* enforces fault tolerance
  - Requires making progress towards a solution

### *Definition 0.* Atomic Commit

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

Based on transaction atomicity from ACID  
(Atomicity, Consistency, Isolation, Durability)

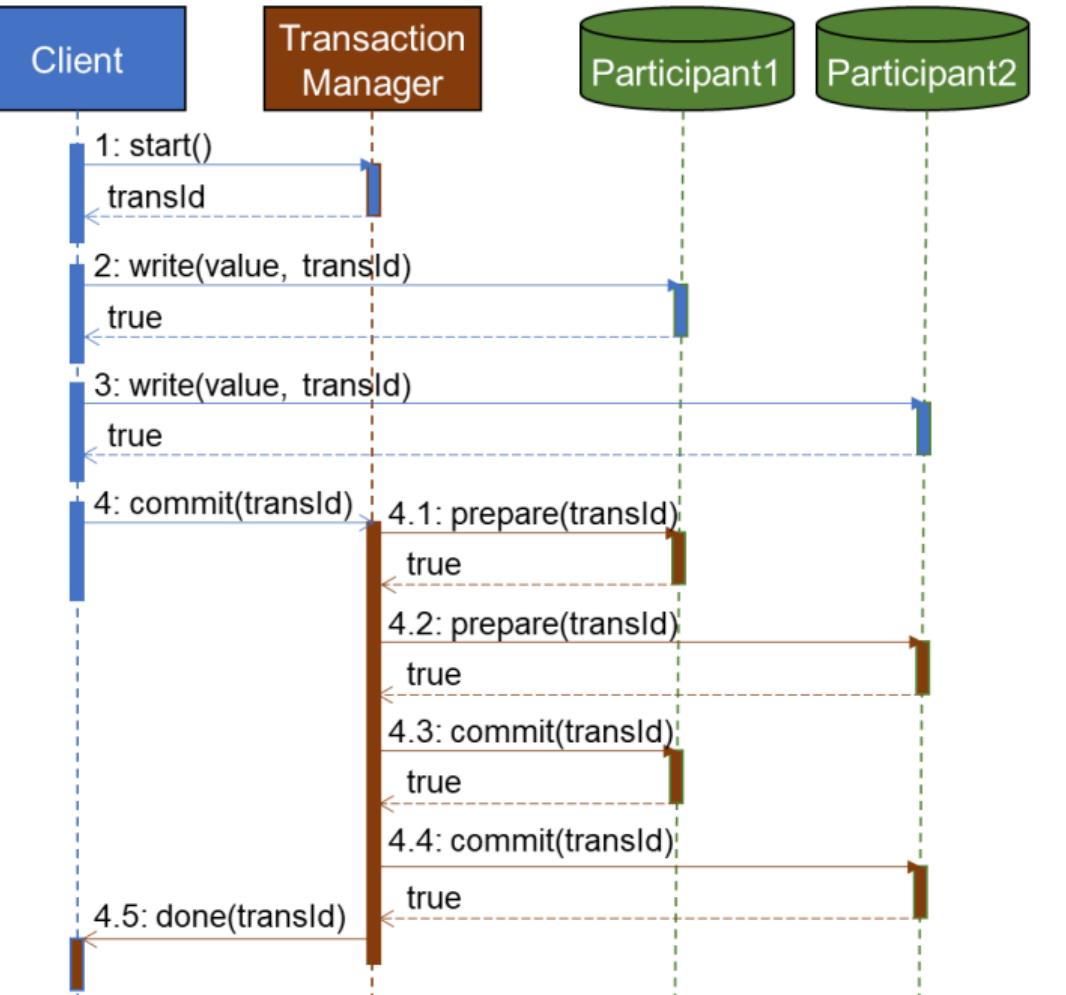
## *Two-Phase Commit*

**Prepare** Confirm nodes can commit transaction

**Commit** Finalise commit once consensus is reached

- Abort if consensus can't be reached

Two-Phase commit example is on next slide.



- Transaction ID used to track writes
- Prepare does all steps of a commit, *aside* from confirming it
  - It cannot be revoked by participant
- Commit intent is recorded in log before sending to participants
- Even if a participant fails, commit can proceed when it recovers
- Comment on *performance costs*

## 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

Cloud-based system that kills crashed nodes.

## 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
  - Any system that allows nodes to be restarted.
  - May lose some steps in memory for *non-critical* tasks.

## 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

Nodes may send faulty, corrupt or malicious messages.

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