CMPU4021 Distributed Systems

Revisions - 3

Introduction to transactions

- A group of operations often represent a unit of “work”.
- Transaction
  - An operation composed of a number of discrete steps.
- Free from interference by operations being performed on behalf of other concurrent clients
- Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

ACID properties of transactions

Atomicity:

- The transaction happens as a single indivisible action. Everything succeeds or else the entire transaction is rolled back. Others do not see intermediate results.

Consistency:

- A transaction takes the system from one consistent state to another consistent state.
  - A transaction cannot leave the database in an inconsistent state.
    - E.g., total amount of money in all accounts must be the same before and after a transfer funds' transaction

Isolated (Serializable)

- Each transaction must be performed without interference from other transactions - there must be no observation by other transactions of a transaction's intermediate effects.
  - If transactions run at the same time, the final result must be the same as if they executed in some serial order.

Durability

- After a transaction has completed successfully, all its effects are saved in permanent storage.

Atomicity of transactions

Two aspects

1. All or nothing:
   - It either completes successfully, and the effects of all of its operations are recorded in the objects, or (if it fails or is aborted) it has no effect at all.
   - Two further aspects of its own:
     - failure atomicity:
       - the effects are atomic even when the server crashes;
     - durability:
       - after a transaction has completed successfully, all its effects are saved in permanent storage.

2. Isolation:
   - Each transaction must be performed without interference from other transactions
     - there must be no observation by other transactions of a transaction's intermediate effects

Transactions

- Transactions are carried out concurrently for higher performance
- Two common problems with transactions
  - Lost update
  - Inconsistent retrieval
- Solution
  - Serial equivalence

Serial equivalence

- A *serially equivalent interleaving* is one in which the combined effect is the same as if the transactions had been done one at a time in some order
  - Does not mean to actually perform one transaction at a time, as this would lead to bad performance

- The same effect means
  - the read operations return the same values
  - the instance variables of the objects have the same values at the end

Aborted transactions

Two problems associated with aborted transactions:

- ‘Dirty reads’
  - A transaction observes a write from a transaction that has not completed yet.
  - An interaction between a *read* operation in one transaction and an earlier *write* operation on the same object
    - by a transaction that then aborts
  - A transaction that committed with a ‘dirty read’ is not recoverable
- ‘Premature writes’
  - interactions between *write* operations on the same object by different transactions, one of which aborts
- Both can occur in serially equivalent executions of transactions

Strict executions of transactions

- Curing premature writes:
  - if a recovery scheme uses ‘before images’
    - write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted
- Strict executions of transactions
  - to avoid both ‘dirty reads’ and ‘premature writes’.
    - delay both read and write operations
  - If both *read* and *write* operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted.
  - Enforces the property of isolation
- *Tentative versions* are used during progress of a transaction
  - objects in tentative versions are stored in volatile memory

Two-phase locking

- Exclusive locks
  - Server locks object it is about to use for a client
  - If a client requests access to an object that is already locked for another clients, the operation is suspended

- Two phase locking
  - Not permitted acquire a new lock after any release
  - Transactions acquire locks in a *growing* phase and release locks in a *shrinking* phase
  - Ensures ***serial equivalence***

Strict Two Phase Locking

- Two Phase Locking
  - Transaction is not allowed any new locks after it has released a lock.

- Strict Two Phase Locking
  - Any locks acquired are not given back until the transaction completed or aborts (ensures durability).
  - Locks must be held until all the objects it updated have been written to permanent storage.

Flat and Nested Transactions

Flat transaction

- Performed atomically on a unit of work

Nested

- Hierarchical

- Transactions may be composed of other transactions.
- Several transactions may be started from within a transaction
  - we have a top-level transaction and subtransactions which may have their own subtransactions.

- To a parent, a subtransaction is atomic with respect to failures and concurrent access. Transactions at the same level can run concurrently but access to common objects is serialised - a subtransaction can fail independently of its parent and other subtransactions; when it aborts, its parent decides what to do, e.g. start another subtransaction or give up.

Distributed Transaction

- Transaction that updates data on two or more systems
- Implemented as a set of sub-transactions
- Challenge
  - Handle machine, software, & network failures while preserving transaction integrity

Distributed transactions

- A *distributed transaction* refers to a flat or nested transaction that accesses objects managed by
  - *Multiple* servers (processes)
  - All servers need to commit or abort a transaction
- Allows for even better performance
  - At the price of increased complexity

Committing Distributed Transactions

- Transactions may process data at more than one server.
  - Problem: any server may fail or disconnect while a commit for transaction T is in progress.
  - They must agree to commit or abort
    - “Log locally, commit globally.”
- The atomicity property of transactions
  - when a distributed transaction comes to an end, either all of its operations are carried out or none of them.

Fault Tolerance

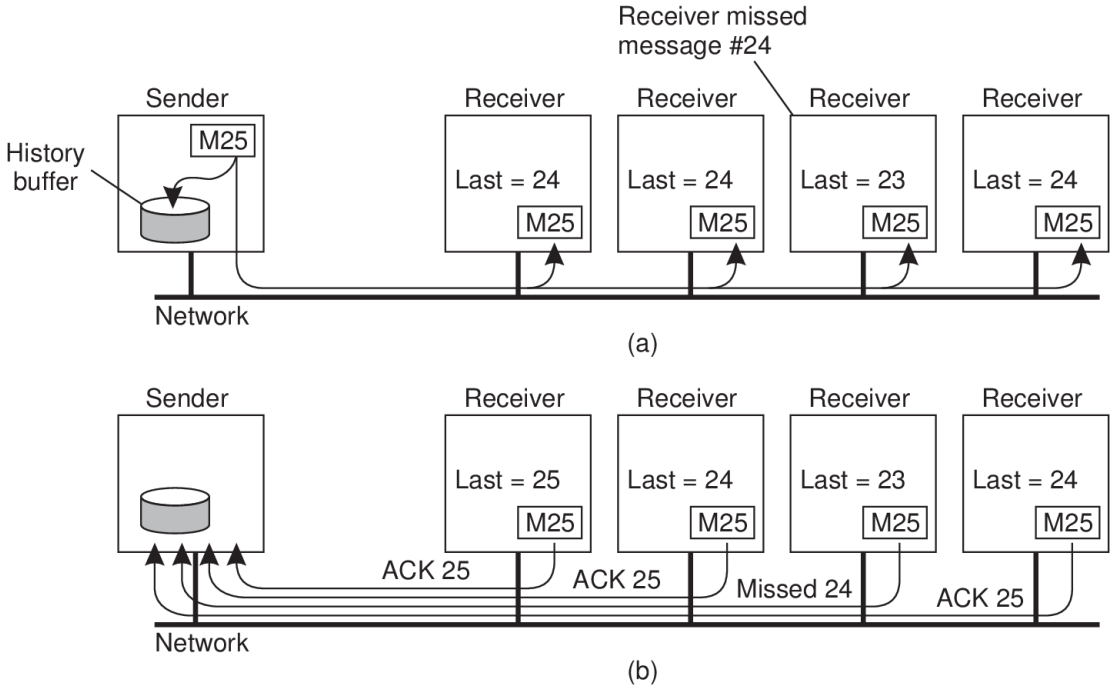
- Distributed systems - the notion of partial failure
  - part of the system is failing while the remaining part continues to operate
  - seemingly correctly

Types of failures

Type	Description of server's behaviour
Crash failure	Halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	Fails to respond to incoming requests Fails to receive incoming messages Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure *Value failure* *State-transition failure*	Response is incorrect The value of the response is wrong. Deviates from the correct flow of control

Simple reliable group communication

- Reliable communication
  - assume nonfaulty processes
- Reliable group communication
  - boils down to **reliable multicasting**:
    - is a message received and delivered to **each** recipient, as intended by the sender.



Atomic multicast

- Problem
  - How to achieve reliable multicasting in the presence of process failures.
- In particular – often needed in a distributed system
  - The guarantee that a message is delivered to either all group members or to none at all.
  - This is also known as the **atomic multicast problem**.
- The atomic multicasting problem is an example of a more general problem, known as **distributed commit**.

Distributed commit protocols

Problem

- Have an operation being performed by each member of a process group, or none at all.
  - Reliable multicasting
    - a message is to be delivered to all recipients.
  - Distributed transaction
    - each local transaction must succeed.

Failure model for the commit protocols

- Commit protocols are designed to work in an asynchronous system in which
  - servers may crash
  - messages may be lost
- It is assumed that an underlying request-reply protocol removes corrupt and duplicated messages.
- There are no Byzantine faults
  - servers either crash or obey the messages they are sent

Atomic commit protocols

- One coordinator and multiple participants
- Protocols for atomic distributed commit
  - One-phase
    - the coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they have carried it out.
  - Two-phase
    - designed to allow any participant to abort its part of a transaction
    - can result in extensive delays for participants in the uncertain state.
  - Three-phase
    - designed to alleviate delays due to participants in the uncertain state.
    - more expensive in terms of the number of messages and the number of rounds
    - required for the normal (failure-free) case.

The two-phase commit protocol

- During the progress of a transaction, the only communication between coordinator and participant is the *join* request
  - The client request to commit or abort goes to the coordinator
    - if client or participant request abort, the coordinator informs the participants immediately
    - if the client asks to commit, the 2PC comes into use
- 2PC
  - *voting phase*: coordinator asks all participants if they can commit
    - if yes, participant records updates in permanent storage and then votes
  - *completion phase*: coordinator tells all participants to commit or abort
  - the next slide shows the operations used in carrying out the protocol

The two-phase commit protocol

Phase 1 (voting phase):

1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
    (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
    (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Performance of the two-phase commit protocol

- If there are no failures, the 2PC involving *N* participants requires
  - *N canCommit?* messages and replies, followed by *N doCommit* messages.
    - the cost in messages is proportional to 3*N*, and the cost in time is three rounds of messages.
    - The *haveCommitted* messages are not counted
- There may be arbitrarily many server and communication failures
- 2PC is guaranteed to complete eventually, but it is not possible to specify a time limit within which it will be complete
  - delays to participants in uncertain state
  - some 3PCs designed to alleviate such delays
    - they require more messages and more rounds for the normal case

Three-phase commit (3PC) protocol: Phase 1

Phase 1: *Voting phase*

- The coordinator sends a `canCommit?` request to each of the participants in the transaction.
  - Purpose: Find out if everyone agrees to commit
- It the coordinator gets a `timeout` from any participant, or any NO replies are received
  - Send an `abort` to all participants
- If a participant times out waiting for a request from the coordinator
  - It `aborts` itself (assume coordinator crashed)
- Else continue to phase 2

3PC protocol: Phase 2

Phase 2: *Prepare to commit phase*

- The coordinator collects the votes and makes a decision.
  - If it is `No`, it `aborts` and informs participants that voted `Yes`
  - if the decision is `Yes`, it sends a `preCommit` request to all the participants.
  - Participants that voted `Yes` wait for a `preCommit` or `doAbort` request.
  - They acknowledge `preCommit` requests and carry out `doAbort` requests.

3PC protocol: Phase 3

Phase 3: *Finalize phase*

- The coordinator collects the acknowledgements.
- When all are received, it commits and sends `doCommit` requests to the participants.
- Participants wait for a `doCommit` request.
- When it arrives, they `commit`.

3PC Weaknesses

- It may result in inconsistent state when a crashed coordinator recovers

- It is not resilient against network partitions
  - Consensus based protocols are designed to be resilient against network partitions
    - Raft, Paxos

Paxos: High Overview

- Paxos is a family of protocols providing distributed consensus
- Goal
  - Agree on a single value even if multiple systems propose different values concurrently
- Common use: provide a consistent ordering of events from multiple clients
  - All machines running the algorithm agree on a proposed value from a client
  - The value will be associated with an event or action
  - Paxos ensures that no other machine associates the value with another event

Consistency, availability, and partitioning (CAP) theorem

In 2000, Eric Brewer posed an important theorem which was later proven to be correct

CAP theorem

Any networked system providing shared data can provide only two of the following three properties:

C: *consistency*, by which a shared and replicated data item appears as a single, up-to-date copy

A: *availability*, by which updates will always be eventually executed

P: Tolerant to the *partitioning* of process group.

Conclusion

In a network subject to communication failures, it is impossible to realize an atomic read/write shared memory that guarantees a response to every request.

CAP Theorem Practical Ramification

- The CAP theorem is all about reaching a trade-off between safety and liveness, based on the observation that obtaining both in an inherently unreliable system cannot be achieved.
  - Practical distributed systems are inherently unreliable.

- One can argue that the CAP theorem moves designers of distributed systems from theoretical solutions to *engineering* solutions.

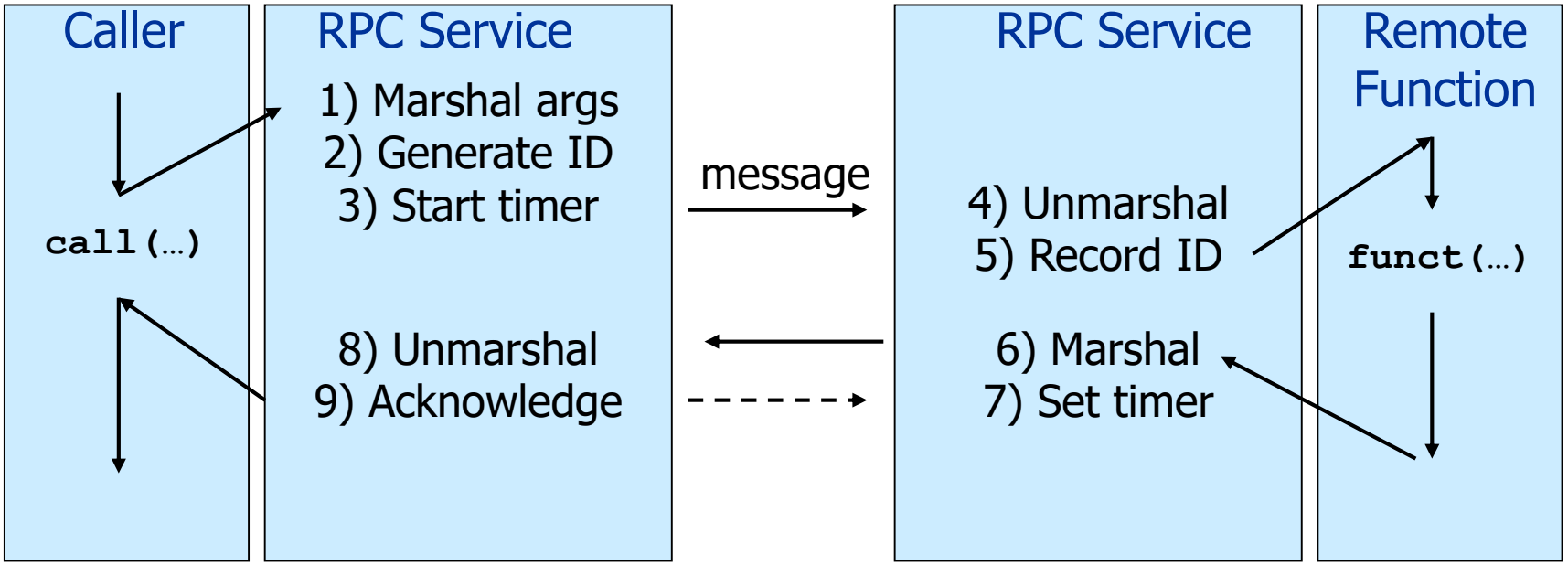
- In practical distributed systems, one simply has to make a choice to proceed despite the fact that another process cannot be reached.
  - In other words, we need to do something when a partition manifests itself through high latency
  - Exactly deciding on how to proceed is application dependent

Middleware - Types

- No agreed classification
- Remote Procedure Call (RPC) middleware
  - Distributed Computing Environment (DCE) RPC - developed the Open Group.
    - It forms the basis for Microsoft's distributed computing environment DCOM
- Object-Oriented Middleware (OOM)
  - Java RMI
  - CORBA
- Message-Oriented Middleware (MOM)
  - Java Message Service
  - IBM MQ, IBM WebSphere
  - Web Services
- Publish/Subscribe Middleware
  - IBM Event Streams, an event-streaming platform built on open-source Apache Kafka technology
- Peer-to-peer middleware
  - Gnutella, Freenet

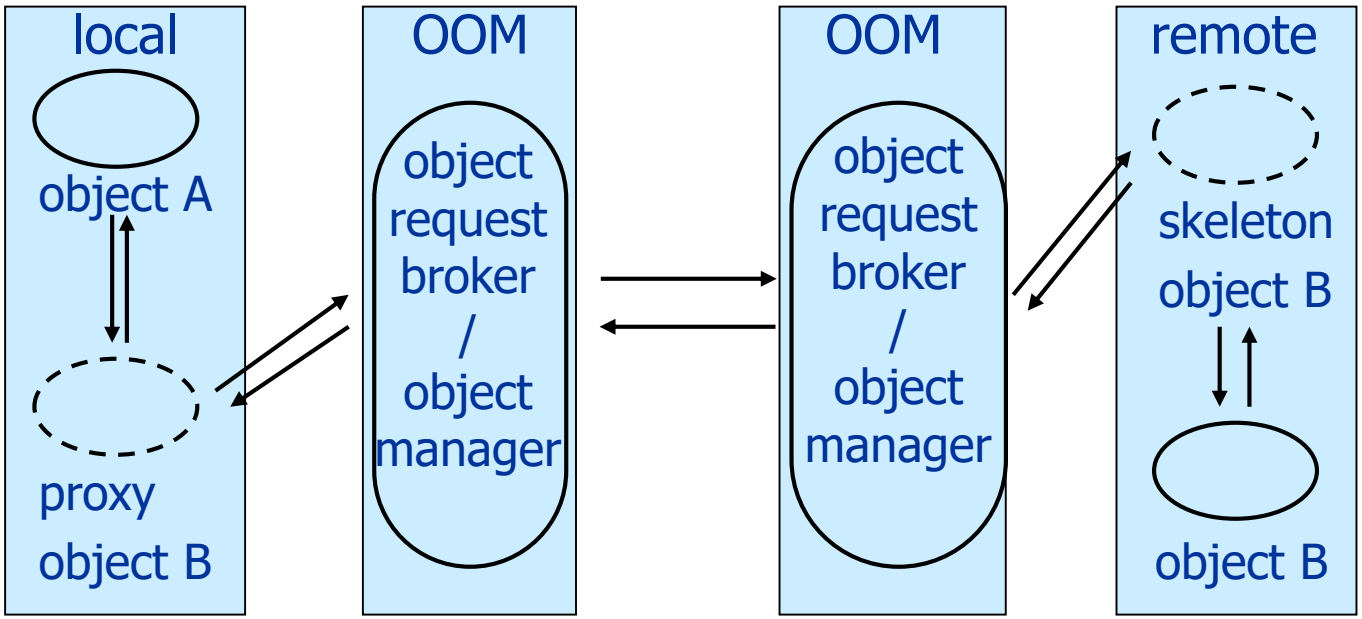
Remote Procedure Call (RPC)

- Allows client programs to call procedures transparently in server programs running in separate processes and in different computers from the client.
- Masks remote function calls as being local
- Request/reply paradigm - message passing
- Provides marshalling of function parameters and return value



Object-Oriented Middleware (OOM)

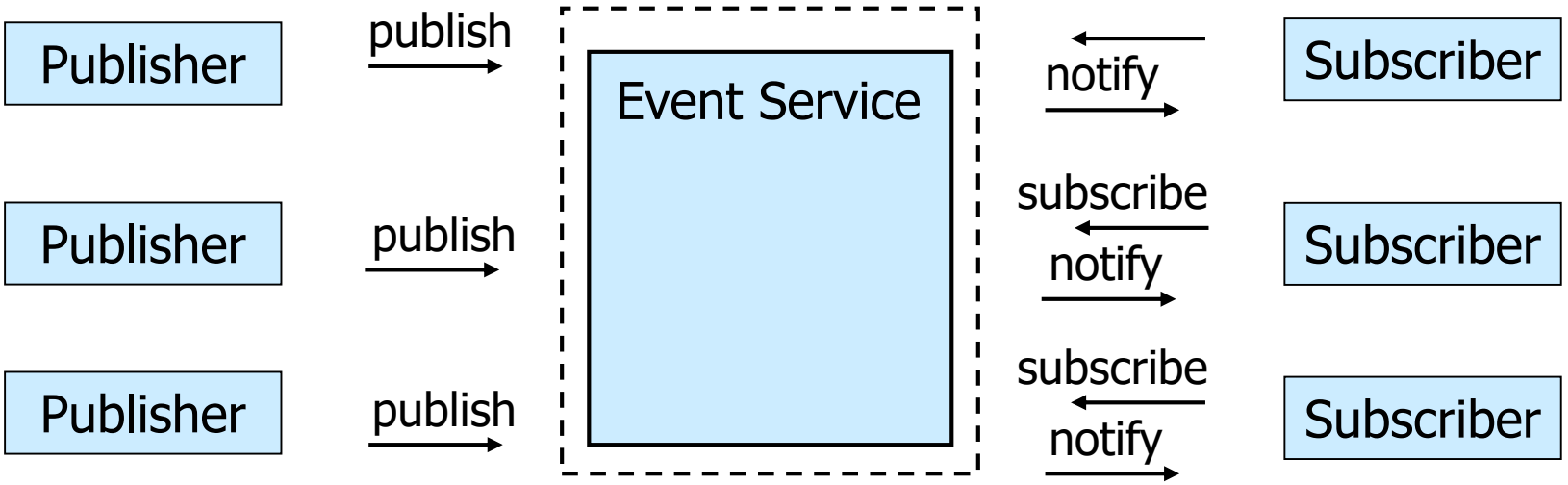
- **Objects** can be *local* or *remote*
- **Object references** can be *local* or *remote*
- Remote objects have visible **remote interfaces**
- Masks remote objects as being local using **proxy objects**
- **Remote method invocation**



https://www.cl.cam.ac.uk/teaching/0910/ConcDistS/

Publish/Subscribe Middleware/

- **Publishers** *publish* **events** (messages)
- **Subscribers** express interest in events with *subscriptions*
- **Event Service** *notifies* interested subscribers of published events
- Events can have arbitrary content or name/value pairs

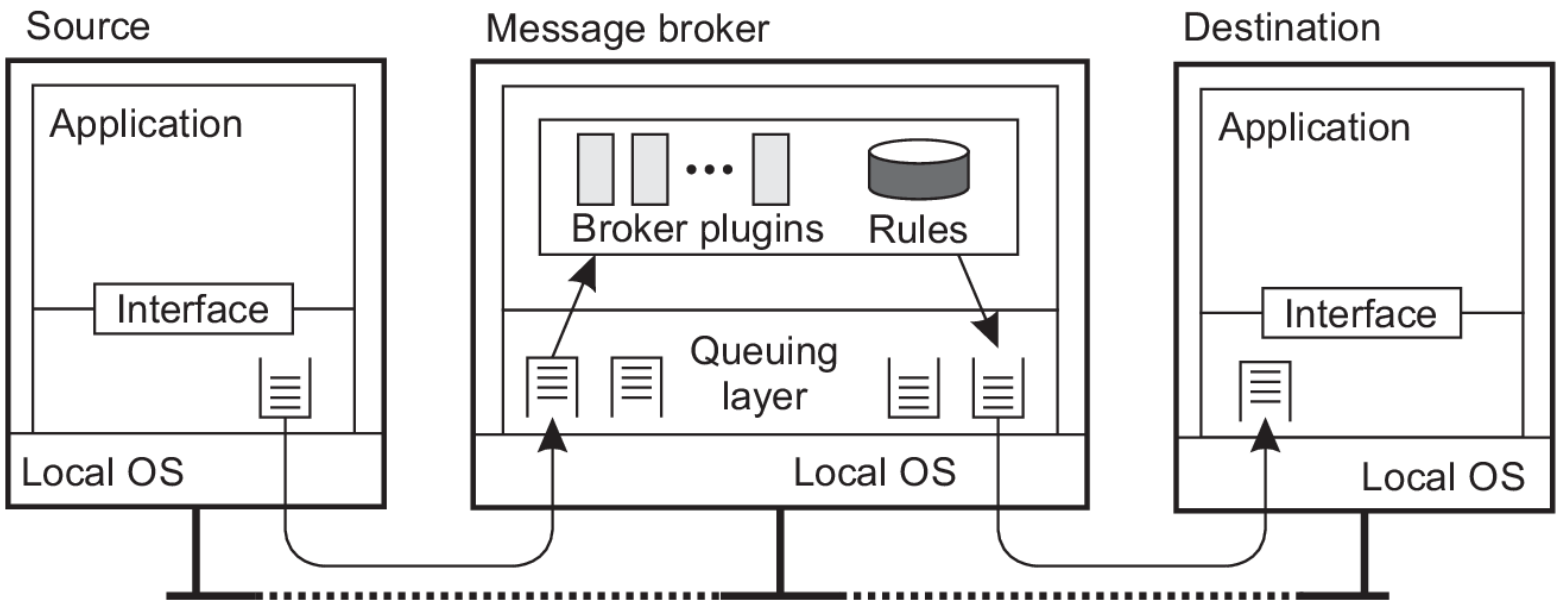


Message Queues aka Message Oriented Middleware (MOM)

- Asynchronous persistent communication through support of middleware-level queues.
- Queues correspond to buffers at communication servers.
- Based on message passing
- Extensive support for persistent asynchronous communication
  - have intermediate-term storage capacity for messages
  - neither sender nor receiver required to be active during transmission
- Not a new idea
  - it is how networks work
    - for example, Unix sockets
- Messages can be large
  - time in minutes
    - as opposed to sockets, where seconds

Message-Oriented Middleware (MOM)

- Communication using **messages**
- Messages stored in **message queues**
- Optional **message server** decouples client and server
- Various assumptions about **message content**



MOM Examples/Toolkits

- A major class of commercial middleware with key implementations including
  - IBM's MQ (previously WebSphere MQ),
  - Microsoft's MSMQ
  - Amazon Simple Queue Service

- Other Examples:
  - Jakarta Messaging
    - a Java Message Oriented Middleware API
    - create, send, and receive messages via loosely coupled, reliable asynchronous communication services.
  - Web Services

- The MOM paradigm has had a long history in distributed applications.
  - Message Queue Services (MQS) have been in use since the 1980's.

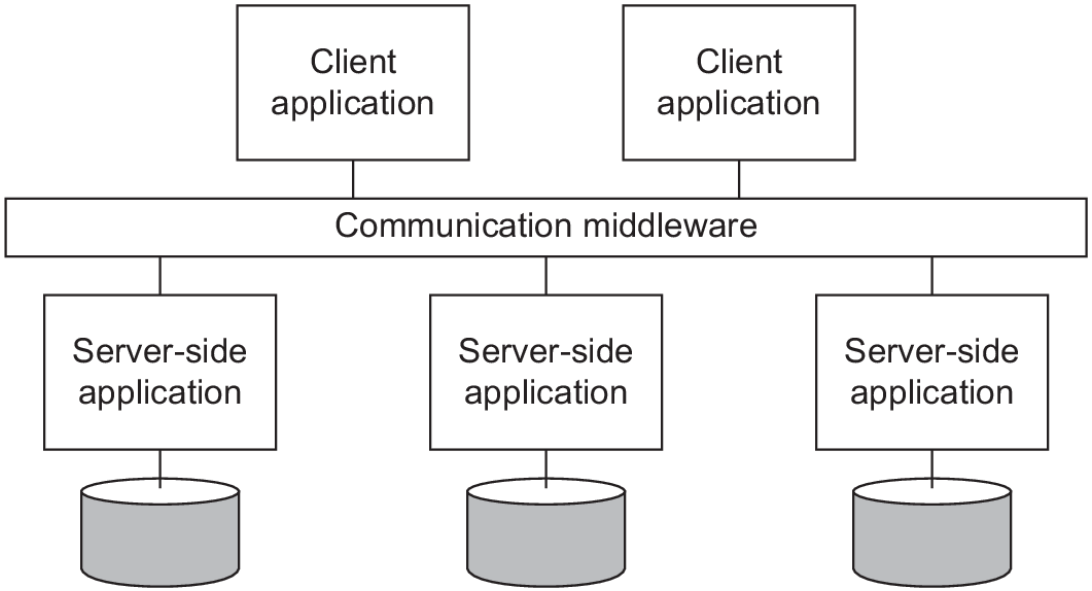
Peer-to-peer middleware

- Peer-to-peer middleware systems are designed specifically to meet the need for
  - the automatic placement and subsequent location of the distributed objects managed by peer-to-peer systems and applications.

Middleware and Enterprise Application Integration (EAI)

- Middleware offers communication facilities for integration

- RPC:
  - Requests are sent through local procedure call, packaged as message, processed, responded through message, and result returned as return from call.
- MOM:
  - Messages are sent to logical contact point (published), and forwarded to subscribed applications.



Middleware and EAI

- Application integration will generally not be simple.
- Middleware - in the form of a distributed system
  - Can significantly help in integration by providing the right facilities such as support for RPCs or messaging.
- Supporting enterprise application integration is an important goal and target field for many middleware products

Criteria for selecting *middleware*

- Suitability
  - integration of software/hardware aspects of architectures
  - Users will only be satisfied if their middleware–OS combination has good performance.
  - Middleware runs on a variety of OS–hardware combinations (platforms) at the nodes of a distributed system.
- Integration of applications
  - standards and middleware technology considerations
- Reliability and robustness
- Transparency
- Risks and cost aspects