CMPU4021 Distributed Systems

Middleware and Distributed Systems

Middleware

- Commonly heard term
- But no generally agreed meaning
- In the context of client server systems
  - Supports communication between clients and servers
    - communication passes through intermediate software layers - the middleware
- In the context of distributed system
  - A separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system

Middleware: Attributes

- Provides services to applications
- Requires system resources, dependencies
- Has vulnerabilities and constraints
- May or may not implement its own access control model
- Developer may not have control over its design
- Provides a consistent programming model

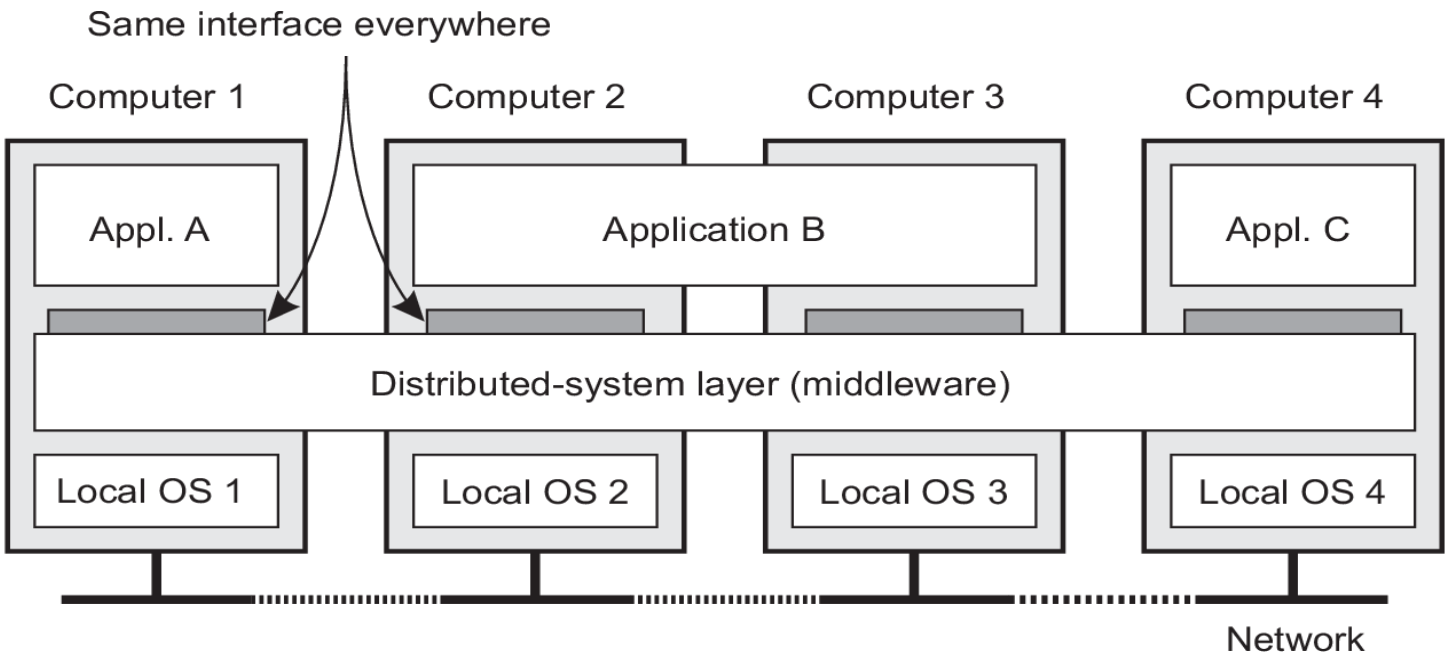
Middleware

- Contains
  - Commonly used components and functions that need NOT be implemented by applications separately.

- The term *middleware* applies to a software layer that provides
  - a programming abstraction as well as
  - masking the heterogeneity of the underlying
    - networks,
    - hardware,
    - operating systems, and
    - programming languages

Distributed systems middleware

- Middleware: the OS of distributed systems
  - A manager of resources offering its applications to efficiently share and deploy those resources across a network.



Middleware layers

Middleware provides:

- *location transparency*:
  - RPC
    - the client that calls a procedure cannot tell whether the procedure runs in the same process or in a different process, different computer.
  - RMI
    - object making the invocation cannot tell whether the object it invokes is local or not;
  - EBP
    - the generating/receiving – not aware of one anothers' locations

- *protocol abstraction*
  - independent of underlying transport protocols

Middleware layers

Provides:

- *OS heterogeneity*
  - independent of the underlying operating system
- *hardware independence*
  - approaches to external data representations hide the differences due to hardware architectures, such as byte ordering.
- *multi-language support*
  - allows clients written in one language to invoke methods in objects that live in server programs written in another language.
    - Achieved by using an interface definition language (IDL) to define interfaces.

Common Middleware Services

- Naming, Location, Service discovery, Replication
- Protocol handling, Communication faults, QoS
- Synchronisation, Concurrency, Transactions, Storage
- Access control, Authentication

Typical middleware services (1)

- Communication
  - E.g. RPC - a developer need only to specify the function header expressed in a special programming language, from which the RPC subsystem can then generate the necessary code that establishes remote invocations.

- Transactions
  - Many applications make use of multiple services that are distributed among several computers.
  - Middleware generally offers special support for executing such services providing *all-or-nothing* feature.
    - The application developer need only specify the remote services involved, and by following a standardized protocol, the middleware makes sure that *every service is invoked, or none at all.*

Typical middleware services (2)

- Service composition
  - Web-based middleware can help by standardizing the way Web services are accessed and providing the means to generate their functions in a specific order.
    - E.g. Web pages that combine and aggregate data from different sources.
    - Well-known mashups are those based on Google maps in which maps are enhanced with extra information such as trip planners or real-time weather forecasts.

- Reliability
  - Providing enhanced functions for building reliable distributed applications, such as
    - E.g. message sent by one process is guaranteed to be received by all or no other process - such guarantees can greatly simplify developing distributed applications

Middleware dimensions

• Request/Reply Messaging	vs.	Asynchronous
• Language-specific	vs.	Language-independent
• Proprietary	vs.	Standards-based
• Small-scale	vs.	Large-scale
• Tightly-coupled components	vs.	Loosely-coupled

Asynchronous Middleware

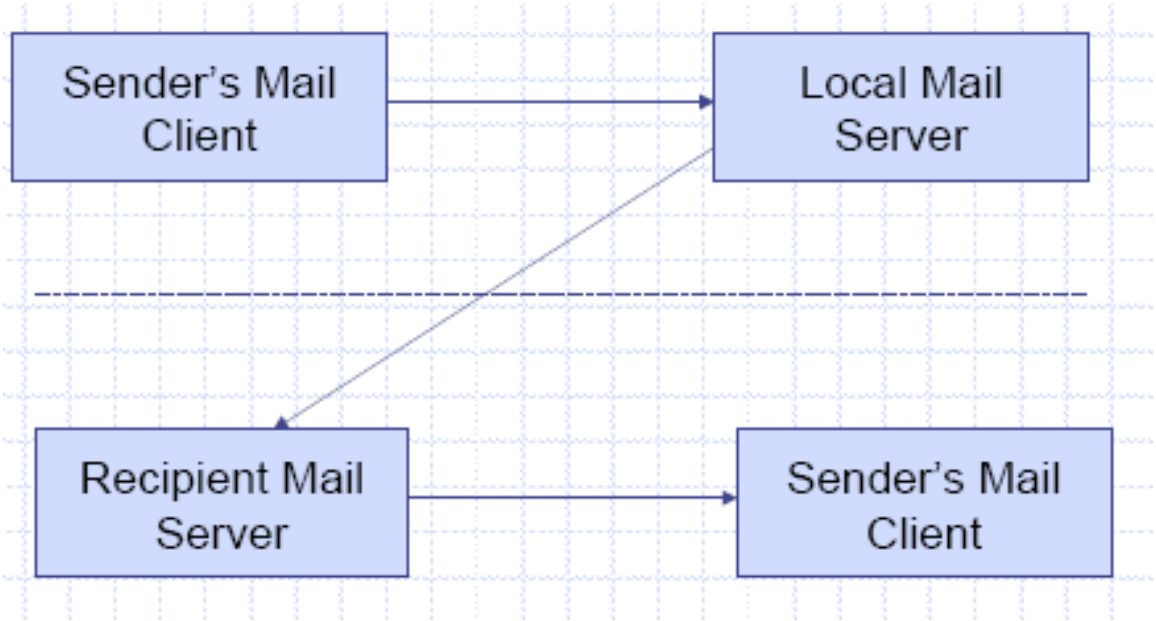
- The client is not assumed to wait for the server after issuing request
- It may continue processing before reply arrives
- Often handled using message passing
  - Message Oriented Middleware (MOM)

Persistent vs. Transient

- Another classification of communication
  - including middleware
- **Persistent**
  - message life does not depend on continued sender execution
- **Transient**
  - message life does depend on continued sender execution

Example

- Electronic mail
  - user message sent to local mail server
  - stored in temporary buffer
  - subsequently sent to target mail server
  - placed in mail box for recipient to read
  - note the places at which communication can be delayed waiting for delivery



Persistent Communication

- Message stored by communication system as long as it takes to deliver
- Sending application does not need to keep executing after sending
- Receiving applications does not have to be executing when message sent
- Better at handling failures (than transient)
  - network failure not a problem
  - other failures can be handled by retry (maybe)

Transient Communication

- Message stored only as long as both sending and receiving application are executing

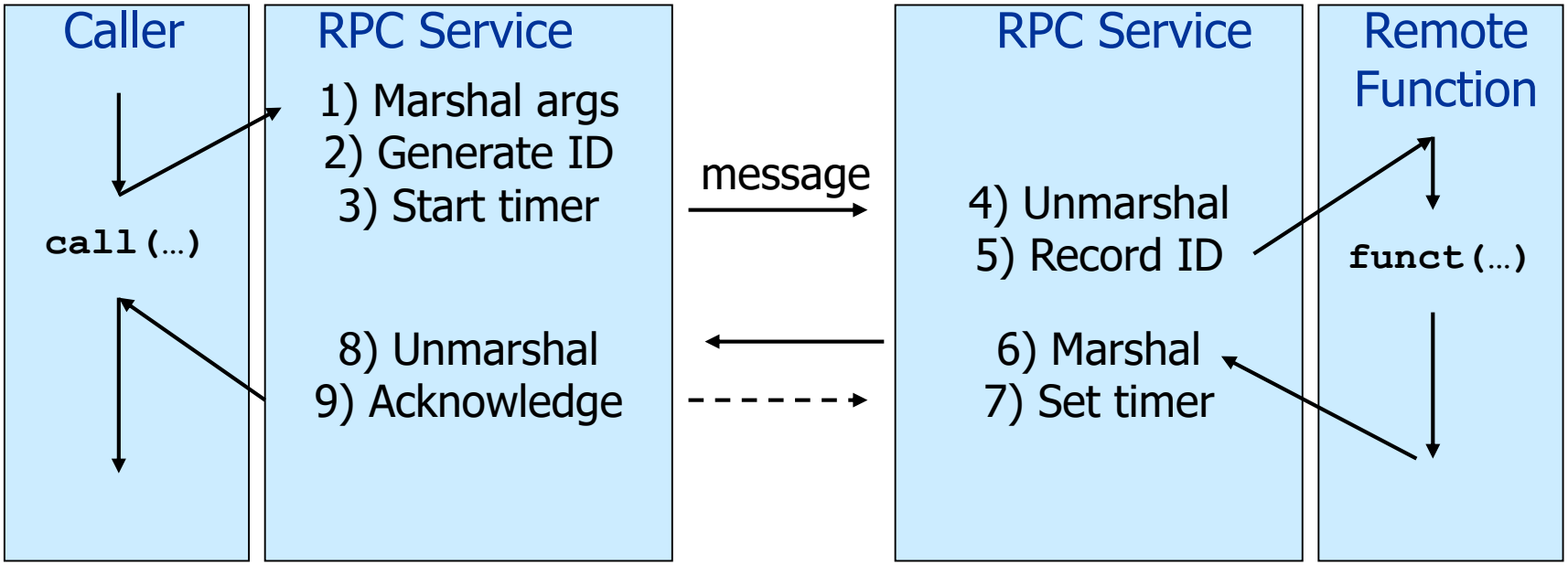
- Can have transient asynchronous
  - both active
  - but sender continues immediately

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



Advantages of RPC

- ✓ Language-level pattern of **function call**
  - easy to understand for programmer
- ✓ **Synchronous request/reply** interaction
  - natural from a programming language point-of-view
  - matches replies to requests
  - built in synchronisation
- ✓ **Distribution transparency** (no-failure case)
  - hides the complexity of a distributed system
- ✓ Various **reliability** guarantees
  - deals with some distributed systems aspects of failure

Disadvantages of RPC

✖ Synchronous request/reply interaction
- tight coupling between client and server
- may block for a long time
- leads to multi-threaded programming

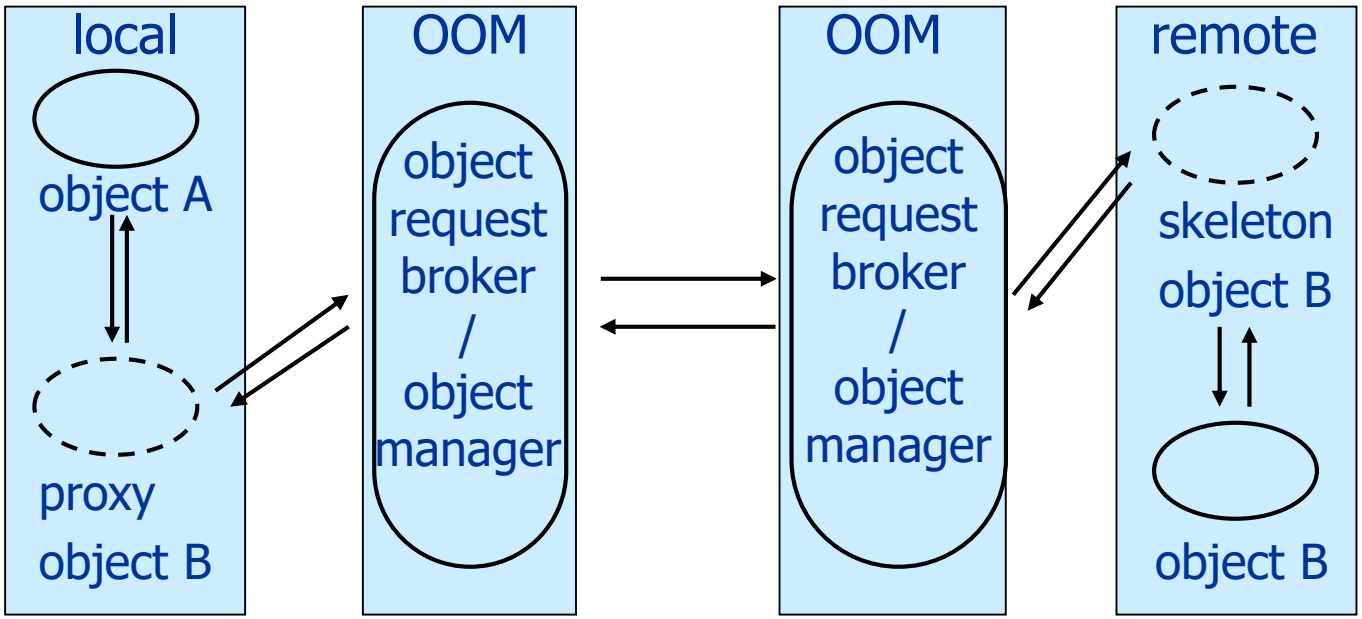
✖ Distribution Transparency
- Not possible to mask all problems

✖ Lacks notion of services
- programmer not interested in server but in service

✖ RPC paradigm is not object-oriented
- invoke methods on objects as opposed to functions on servers

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/

Properties of OOM

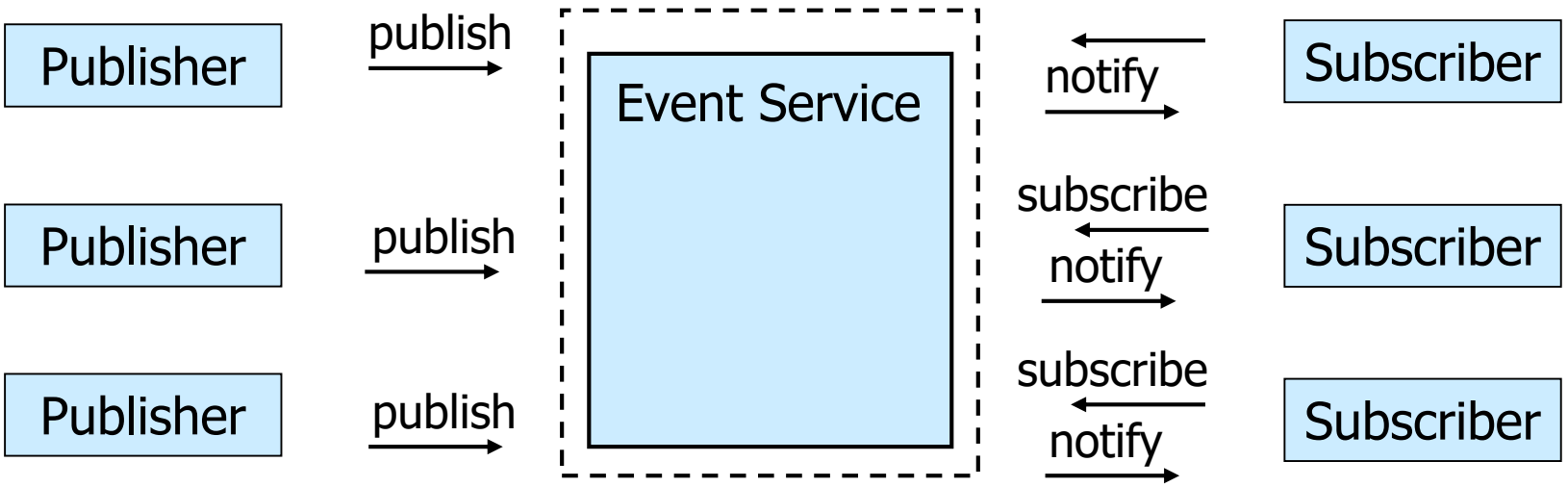
- ✓ Support for object-oriented programming model
  - objects, methods, interfaces, encapsulation, …
  - exceptions (also in some RPC systems)
- ✓ **Location Transparency**
  - mapping object references to locations
- ✓ Synchronous request/reply interaction
  - same as RPC
- ✓ Services comprising multiple servers are easier to build with OOM

Disadvantages of OOM

- ✖ Synchronous request/reply interaction
  - Asynchronous Method Invocation (AMI)
  - But implementations may not be loosely coupled
- ✖ Distributed garbage collection
  - Releasing memory for unused remote objects
- ✖ OOM rather static and heavy-weight
  - Bad for ubiquitous systems and embedded devices

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



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

Kafka

- Apache Kafka
  - Process streams of records in real time – huge amunts
- Open source publish-subscribe messaging system
  - Allows consumers to subscribe to topics for which they want to receive messages
  - Developed by LinkedIn in 2011 - open-sourced and donated Kafka to the Apache
  - Written in Scala and Java
- The main functions:
  - Enables applications to publish or subscribe to data or event streams.
  - Stores records in the order in which they occurred
    - in a fault-tolerant and durable way.
  - It processes records in real-time - as they occur.

Apache Kafka

- Distributed
  - Each message queue is divided among multiple servers
- Scalable
  - Splitting a topic into multiple partitions across many systems - servers
- Durable
  - Store streams of records in the order in which records were generated
  - Records written to disk using large streaming writes
- Fault tolerant
  - Messages are written to disk for a configurable time period
  - Partitions can be replicated onto multiple servers
    - a leader & followers per partition also provide support for redundancy

Kafka

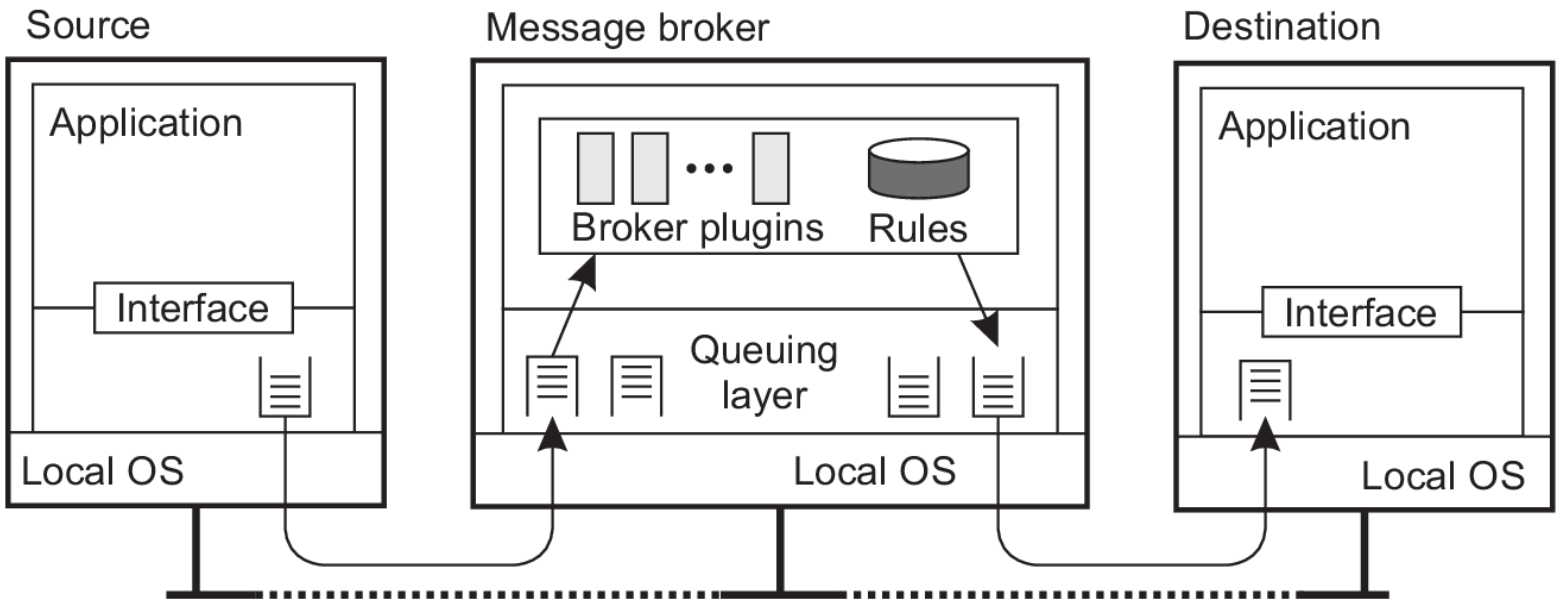
- Runs as a cluster on one or more servers
- Each server is called a broker
  - Ranges from 1 to 1000s of brokers
- Kafka usage
  - Real-time systems, such as Spark Streaming
  - Batch processing
    - For example store to Amazon S3 & then use MapReduce
  - Netflix
  - LinkedIn
  - Uber
  - Tinder
  - AirBnB
  - Apache technologies (e.g. Apache Hadoop)

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**



Properties of MOM

- ✓ **Asynchronous** interaction
  - Client and server are only **loosely coupled**
  - Messages are queued
  - Good for application integration
- ✓ Support for **reliable** delivery service
  - Keep queues in persistent storage
- ✓ Processing of messages by intermediate message server
  - Filtering, transforming, logging, …
  - Networks of message servers
- ✓ Natural for database integration

Disadvantages of MOM

- ✖ Poor programming abstracting
  - Rather low-level (cf. Packets)
  - Results in multi-threaded code
  - Request/reply more difficult to achieve
- ✖ Message formats unknown to middleware
  - No type checking
- ✖ Queue abstraction only gives one-to-one communication
  - Limits scalability

MOM/MQ - additional functionalities

- Transactions support
  - Support for the sending or receiving of a message to be contained within a transaction
- Message transformation
  - An arbitrary transformation can be performed on an arriving message.
    - E.g. - to transform messages between formats to deal with heterogeneity in underlying data representations.
  - Important tool in dealing with heterogeneity
  - Message broker
    - Term often used to denote a service responsible for message transformation.

Message queues vs Message Passing

- Message queues are similar to the message-passing systems
- The difference
  - message-passing systems have *implicit* queues associated with senders and receivers
  - message queuing systems have *explicit* queues that are third-party entities, *separate* from the sender and the receiver.
- This is the key difference that makes message queues
  - an indirect communication paradigm
  - with the crucial properties
    - of space and time uncoupling.

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.

IBM's WebSphere MQ

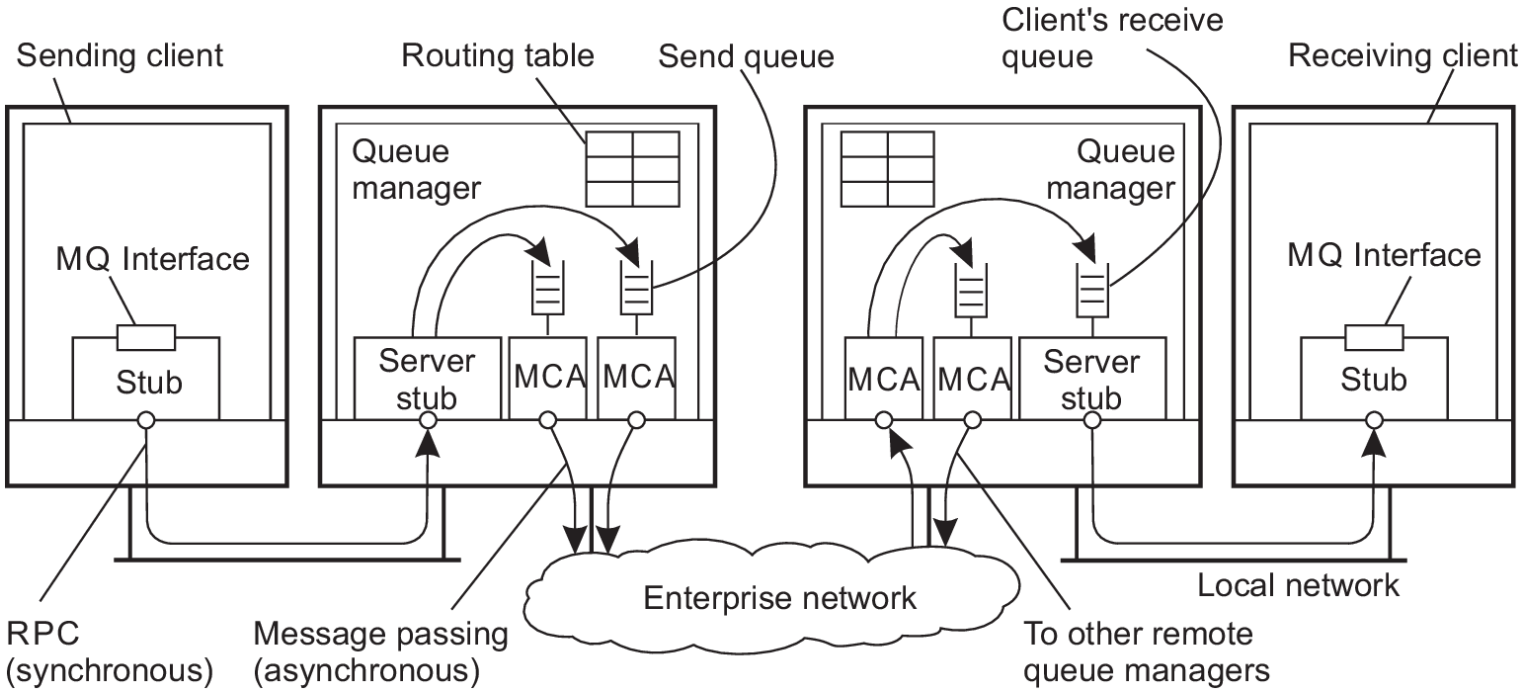
- IBM messaging middleware - application platform

- Basic concepts
  - **Application-specific messages** are put into, and removed from **queues**
  - Queues reside under the regime of a **queue manager**
  - Processes can put messages only in local queues, or through an RPC mechanism

- Message transfer
  - Messages are transferred between queues
  - Message transfer between queues at different processes, requires a **channel**
  - At each end point of channel is a **message channel agent**
  - Message channel agents are responsible for:
    - Setting up channels using lower-level network communication facilities (e.g., TCP/IP)
    - (Un)wrapping messages from/in transport-level packets
    - Sending/receiving packets

IBM's WebSphere MQ: Schematic overview

- Channels are inherently unidirectional
- Automatically start MCAs when messages arrive
- Any network of queue managers can be created
- Routes are set up manually (system administration)



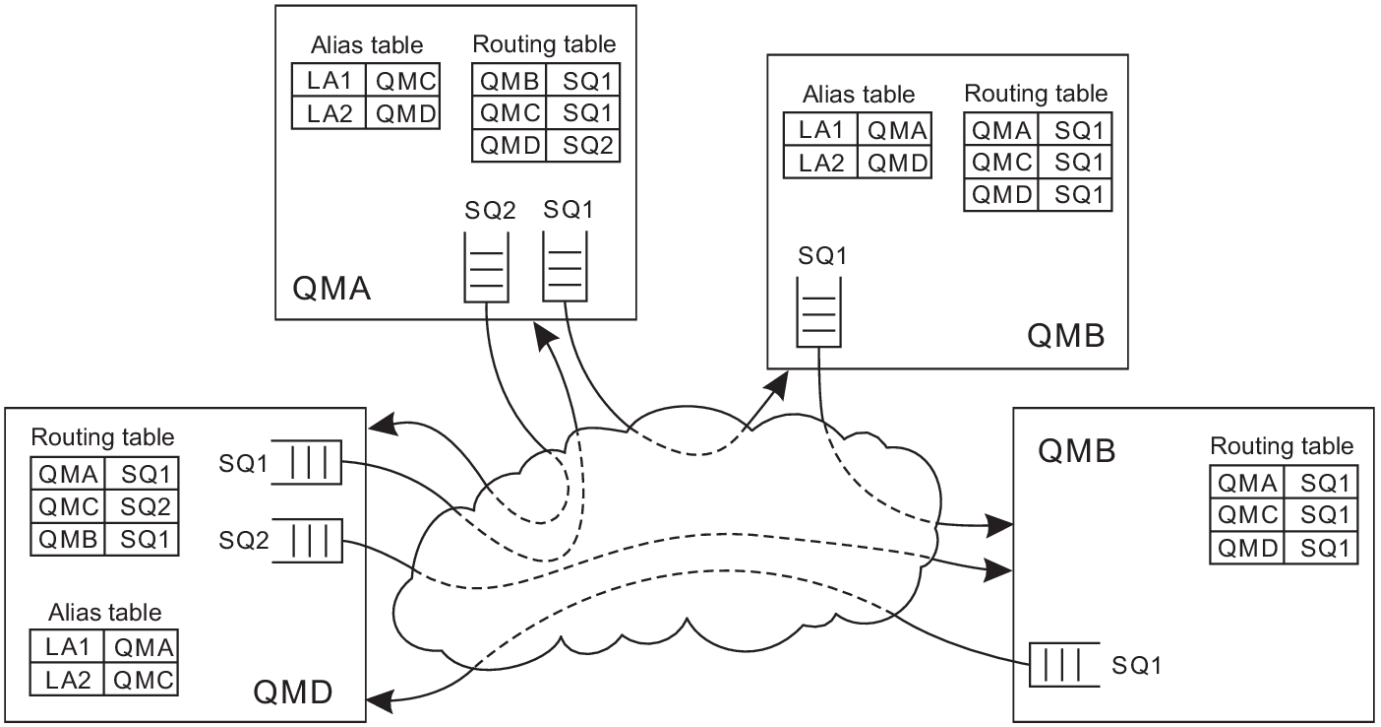
IBM's WebSphere MQ: Message channel agents

Some attributes associated with message channel agents

Attribute	Description
Transport type	Determines the transport protocol to be used
FIFO delivery	Indicates that messages are to be delivered in the order they are sent
Message length	Maximum length of a single message
Setup retry count	Specifies maximum number of retries to start up the remote MCA
Delivery retries	Maximum times MCA will try to put received message into queue

IBM's WebSphere MQ: Routing

- By using logical names, in combination with name resolution to local queues, it is possible to put a message in a remote queue



IBM MQ

- Provides messaging and queuing capabilities for enterprise messaging

- Messaging
  - Programs communicate by sending each other data in messages rather than by calling each other directly.

- Queuing
  - Messages are placed on queues, so that programs can run independently of each other, at different speeds and times, in different locations, and without having a direct connection between them.

- Multiple modes of operation:
  - point-to-point
  - publish/subscribe

- Point-to-point
  - Applications send messages to a queue and receive messages from a queue. Each message is consumed by a single instance of an application. The sender must know the name of the destination, but not where it is.

- Publish/subscribe
  - Applications subscribe to topics. When an application publishes a message on a topic, IBM MQ sends copies of the message to those subscribing applications. The publisher does not know the names of subscribers, or where they are.

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.

Peer-to-peer middleware functions

- To simplify the construction of services that are implemented across many hosts in a widely distributed network.
  - It must enable clients to locate and communicate with any individual resource made available to a service, even though the resources are widely distributed amongst the hosts.

- To have the ability
  - to add new resources and to remove them at will, and
  - to add hosts to the service and remove them.

Non-functional requirements of peer-to-peer middleware

Peer-to-peer middleware must address the following non-functional requirements:

- Global scalability
- Load balancing
- Optimization for local interactions between neighboring peers
- Accommodating to highly dynamic host availability
- Security of data in an environment with heterogeneous trust
- Anonymity, deniability and resistance to censorship

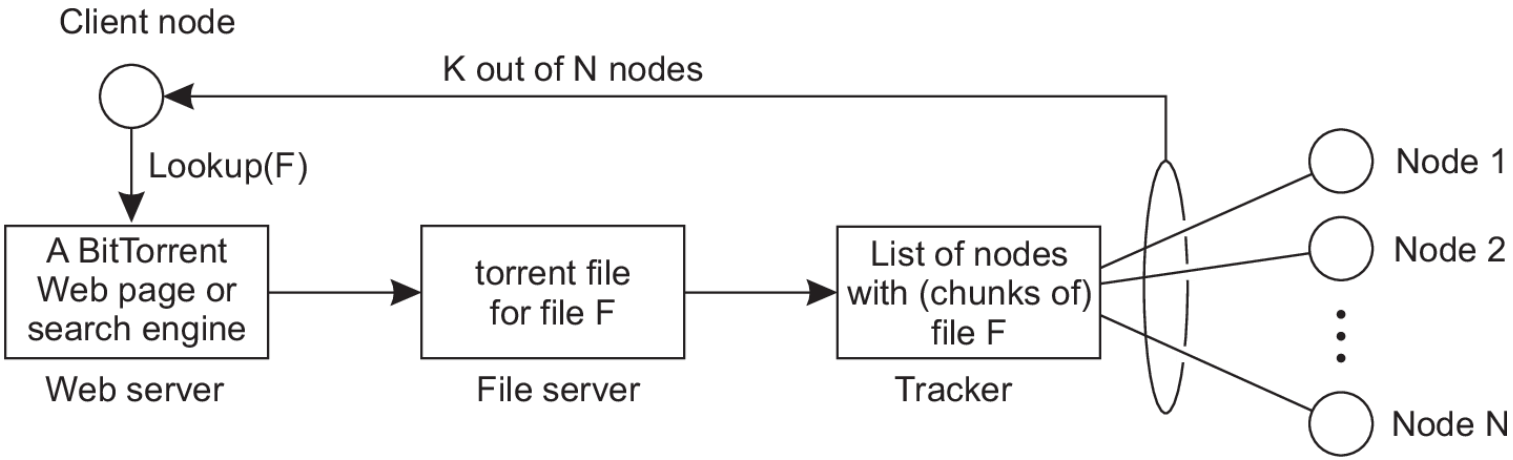
P2P: BitTorrent

- Peer-to-peer file downloading system.
- An important design goal was to ensure collaboration.
- When an end user is looking for a file,
  - downloads chunks of the file from other users
    - until the downloaded chunks can be assembled together - yielding the complete file.

Collaboration: The BitTorrent case

Principle: search for a file F

- Lookup file at a global directory ⇒ returns a **torrent file**
- Torrent file contains reference to **tracker**
  - a server keeping an accurate account of **active** nodes that have (chunks of) F .
- Node P can join **swarm**, get a chunk for free, and then trade a copy of that chunk for another one with a peer Q also in the swarm.



Integrating applications

- Situation
  - Organizations confronted with many networked applications, but achieving interoperability was painful.
- Basic approach
  - A networked application is one that runs on a server making its services available to remote clients.
  - Simple integration:
    - clients combine requests for (different) applications; send that off; collect responses, and present a coherent result to the user.
- Next step
  - Allow direct application-to-application communication, leading to Enterprise Application Integration (EAI).

How to integrate applications

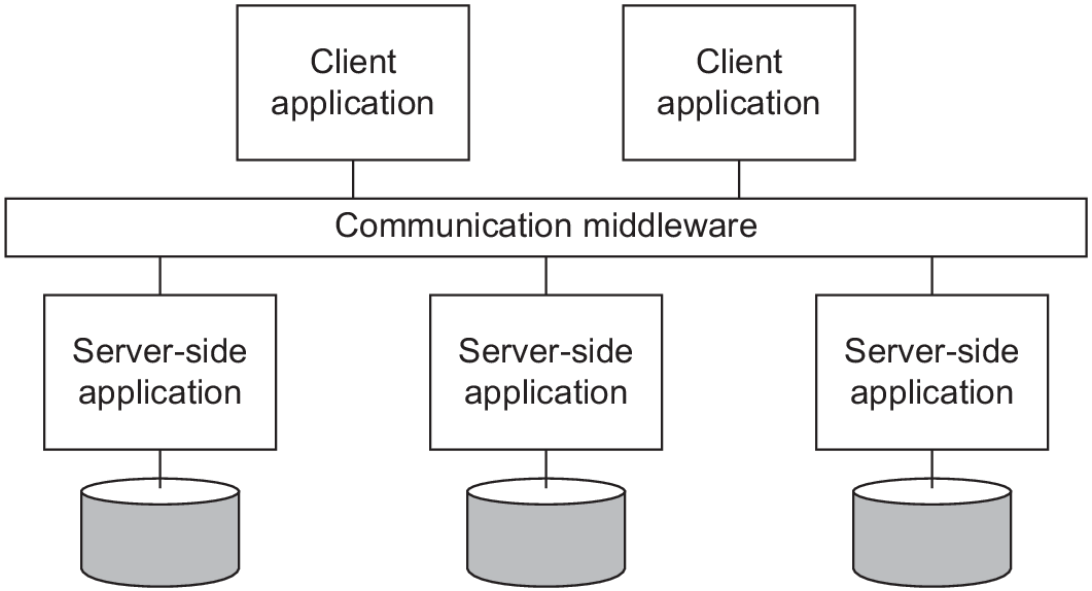
Four basic approaches:

- File transfer
  - Technically simple, but not flexible
    - Figure out file format and layout
    - Figure out file management
    - Update propagation, and update notifications.
- Shared database
  - Much more flexible, but still requires common data scheme next to risk of bottleneck.
- Remote procedure call
  - Effective when execution of a series of actions is needed.
- Messaging
  - RPCs require caller and callee to be up and running at the same time.
  - Messaging allows decoupling in time and space.

Middleware and 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

Criteria for selecting *middleware*

- Strength of product support
  - The maturity and stability of the tool;
  - The fault tolerance provided by the tool;
  - The availability of developer tools;
  - Maintainability;
  - Code reuse
- Security characteristics

Middleware: Security Goals

- Engineer application to depend on middleware only as much as necessary, in view of middleware's capabilities, liabilities and constraints

- Engineer system to account for middleware's capabilities, liabilities and constraints.

Middleware – limitations

- Many distributed applications rely entirely on the services provided by the available middleware to support their needs for communication and data sharing.
  - E.g., systems constrained by use of only RMI for communication and data sharing – consider a C-S model for request/reply of names and addresses in a database.
- Abstraction and full ‘independence’ of application layer is not achieved, yet!
  - E.g., consider a C-S-based mail service provider where a server must add its own error detection/recovery mechanism for retransmission if a link breaks on large transmissions. Or vice-versa, is possible.
- Arguably, error detection/recovery needs to be at several levels, not only at middleware

References

- Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, Section 5.1

- Chapter 4: Maarten van Steen, Andrew S. Tanenbaum, Distributed Systems, 3rd edition (2017)

- Dr J. Bacon, University of Cambridge,
- P. Krzyzanowski, Rutgers, The State University of New Jersey
- https://www.ibm.com/cloud/learn/middleware
- https://www.ibm.com/cloud/learn/apache-kafka