CMPU4021 Distributed Systems

Middleware - Overview

Middleware

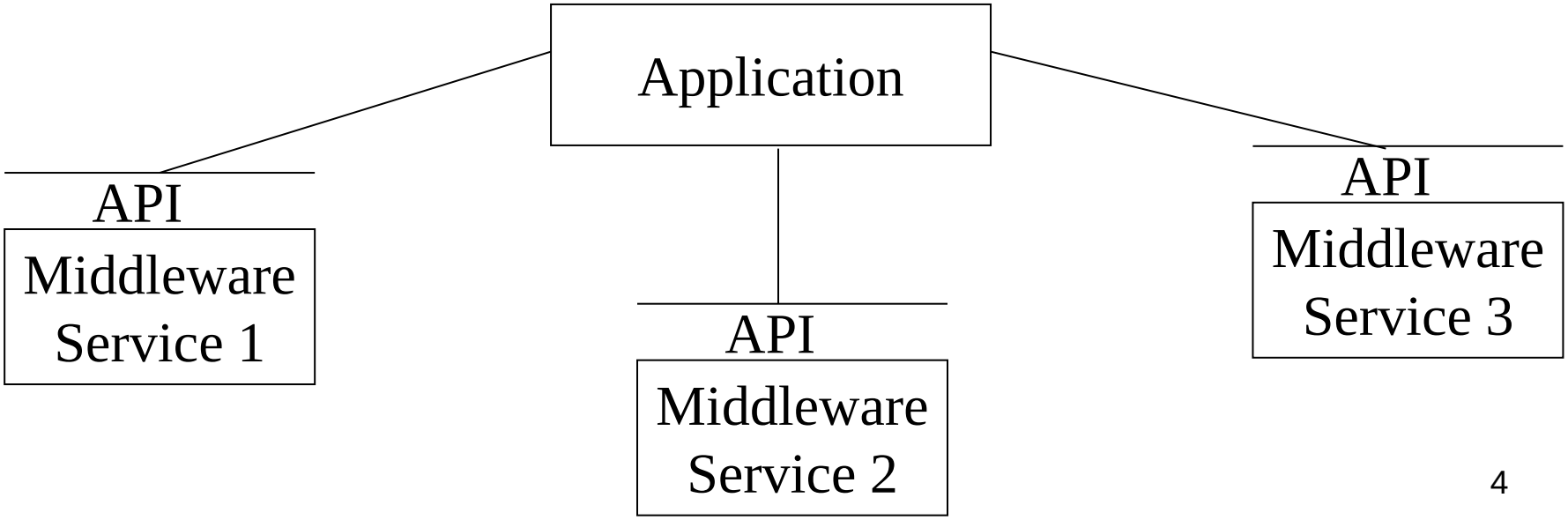
- Commonly heard term
- But no generally agreed meaning
- Most often used in the context of client server systems
  - Clients and servers communicate
  - This communicate often passes through intermediate software layers
  - This intermediate software is the middleware

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

Distributed Systems Middleware

- Enables the modular interconnection of distributed software
  - Usually via services



Common Middleware Services

- Naming and Directory Service
- Event Service
- Transaction Service
- Fault Detection Service
- Trading Service
- Replication Service
- Migration Service

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

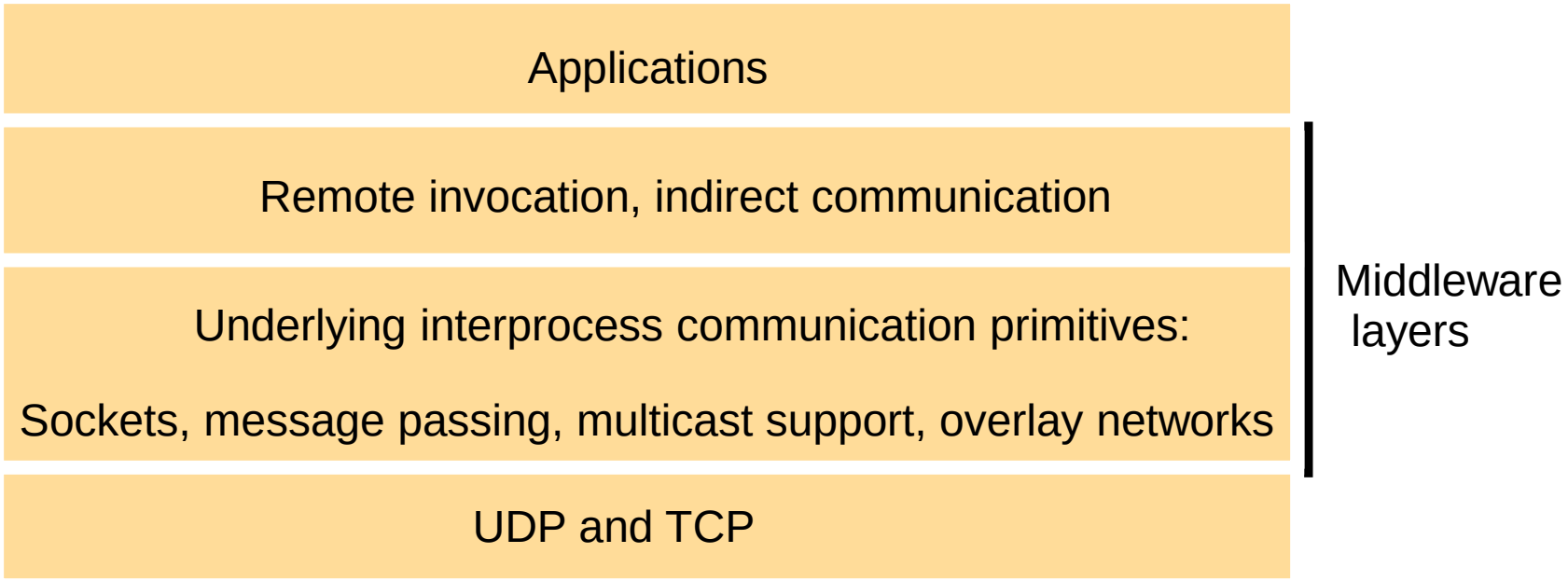
Middleware

- A whole range of technologies can be classified as middleware

- These include
  - RPC
  - Distributed objects and remote method invocation *(Java RMI, Corba)*
  - Remote event notification *(Jini)*
  - Remote SQL access *(JDBC)*
  - Distributed transaction processing

Middleware

- A suite of API software that uses underlying processes and communication (message passing) protocols to provide its abstract protocol – simple RMI request-reply protocol



Middleware

- Supports interaction between clients and servers
- Masks heterogeneity
- Provides a consistent programming model

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.

Middleware

- Middleware provides support for (some of):
  - Naming, Location, Service discovery, Replication
  - Protocol handling, Communication faults, QoS
  - Synchronisation, Concurrency, Transactions, Storage
  - Access control, Authentication

Asynchronous Middleware

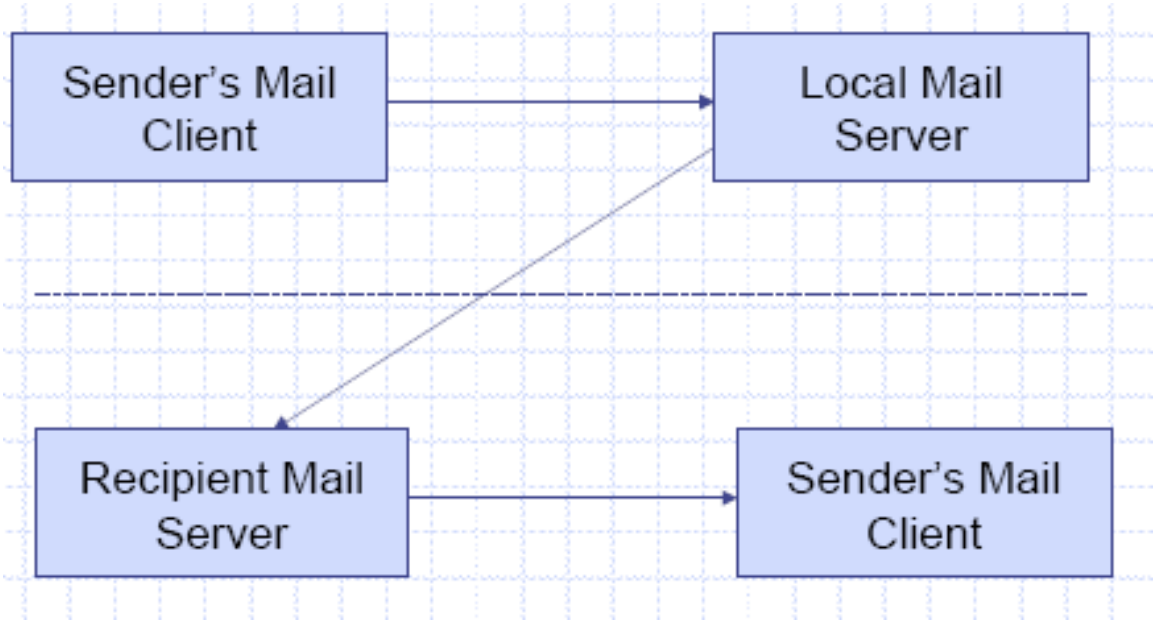
- The client is not assumed to wait for the server after issuing request
- It may continue processing before reply arrives
- Often handled using message passing
  - hence, Message Oriented Middleware (MOM)

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

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

Failure

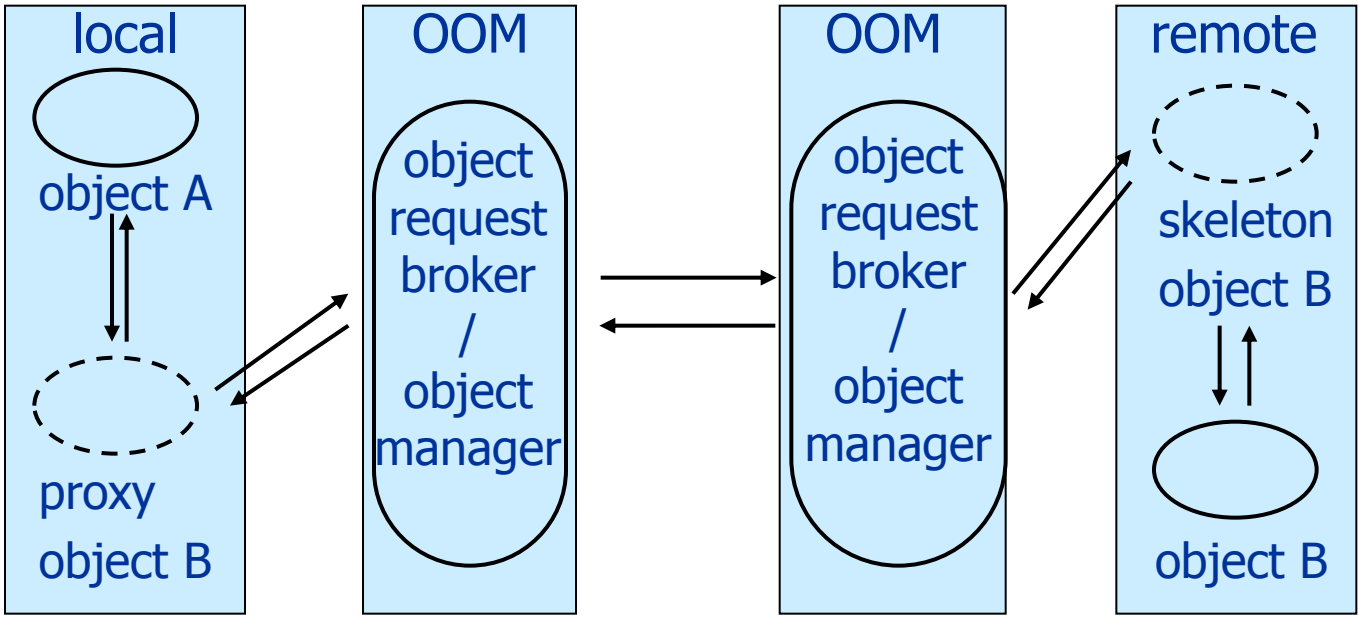
- Persistent communication better at handling failures
  - network failure not a problem
  - other failures can be handled by retry (maybe)

Middleware - Types

- Remote Procedure Call (RPC)
- Object-Oriented Middleware (OOM)
  - Java RMI
  - CORBA
  - Reflective Middleware
    - Flexible middleware (OOM) for mobile and context-aware applications – adaptation to context
- Message-Oriented Middleware (MOM)
  - Java Message Service
  - IBM MQ
  - Web Services
- Event-Based Middleware
  - Jini

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**



Properties of OOM

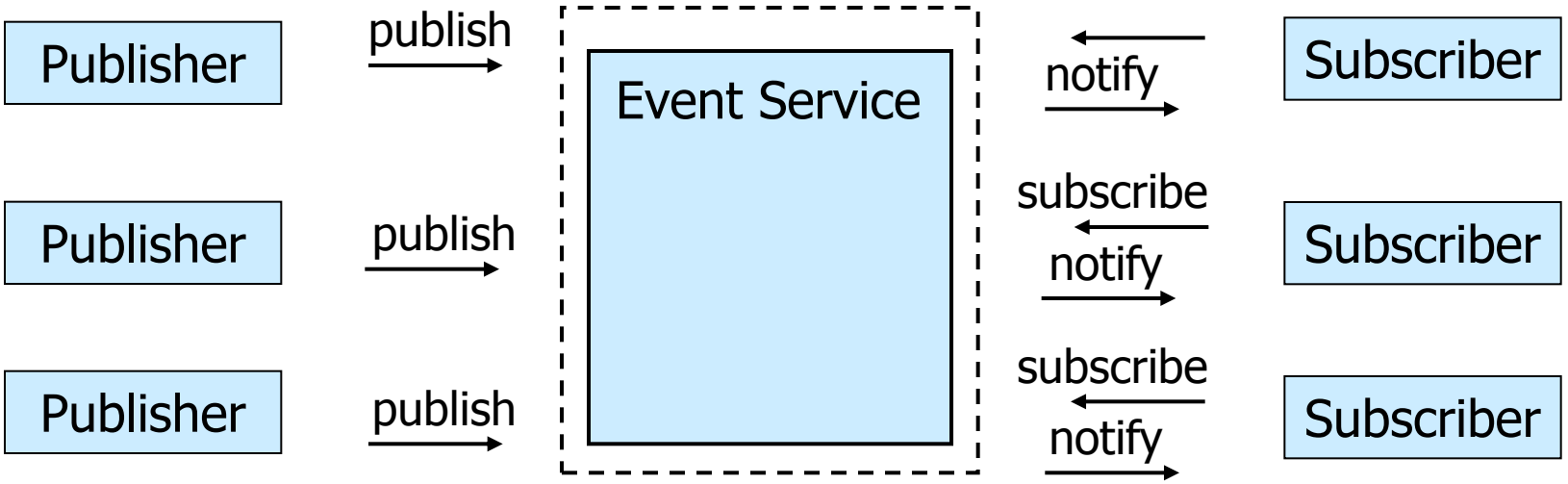
- ✓ Support for object-oriented programming model
  - objects, methods, interfaces, encapsulation, …
  - exceptions (also in some RPC systems)
- ✓ **Location Transparency**
  - mapping object references to locations
- ✓ Synchronous request/reply interaction
  - same as RPC
- ✓ Services comprising multiple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

Event-Based Middleware/ Publish/Subscribe

- **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

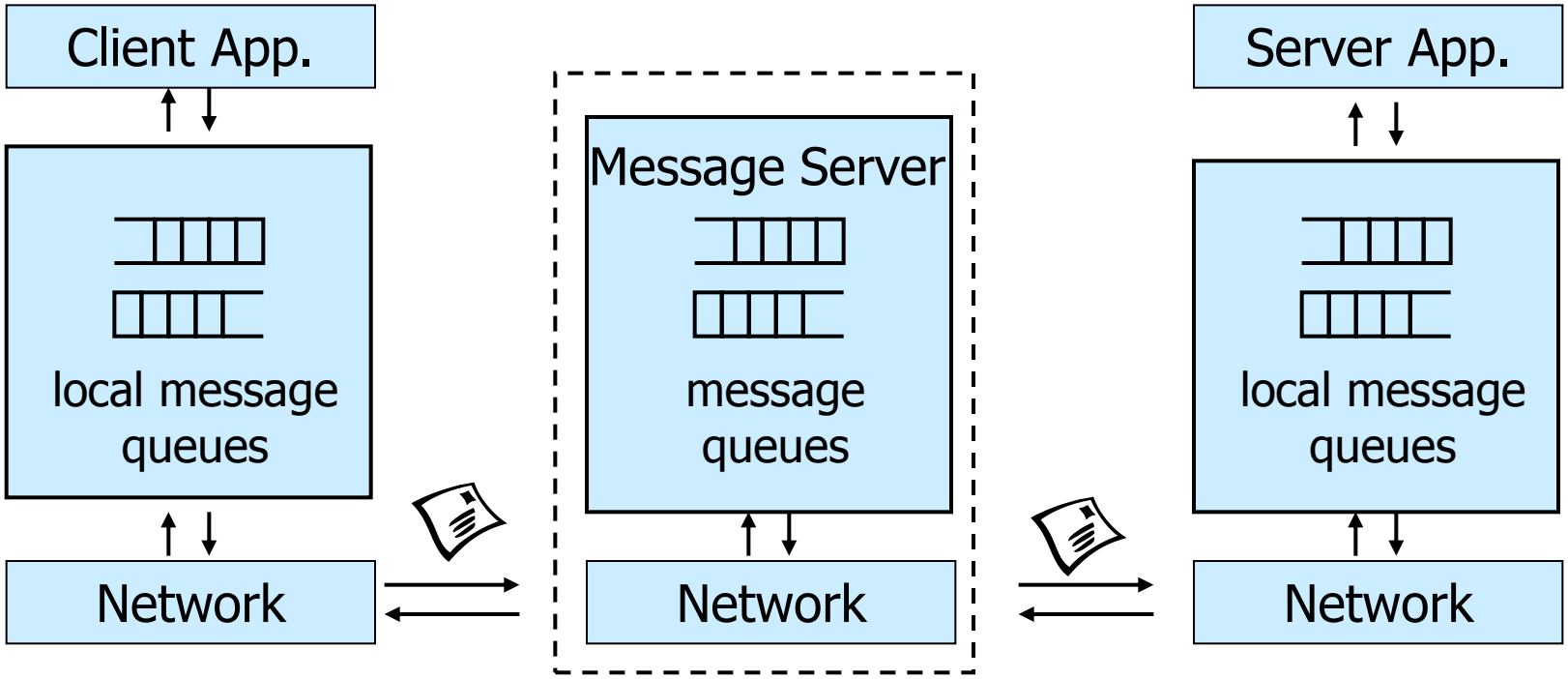


Message Queues aka Message Oriented Middleware (MOM)

- 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
  - Oracle's Streams Advanced Queuing (AQ).

- Other Examples:
  - Java Message Service
  - 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 MQ

- IBM messaging middleware.

- Provides messaging and queuing capabilities across multiple modes of operation:
  - point-to-point
  - publish/subscribe

IBM MQ

Provides:

- 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.

- 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.

Java Message Service (JMS)

- A specification of a standardized way for distributed Java programs to communicate indirectly
  - API specification to access MOM implementations

- Two modes of operation:
  - Point-to-point
    - one-to-one communication using queues
  - Publish/Subscribe
    - As an Event-Based Middleware

- Implementations
  - Joram from OW2,
  - Apache ActiveMQ
  - IBM MQ, also provide a JMS interface on to their underlying infrastructure.

Key roles in JMS

- A JMS client
  - a Java program or component that produces or consumes messages
- JMS producer
  - a program that creates and produces messages
- JMS consumer
  - a program that receives and consumes messages.
- JMS provider
  - any of the multiple systems that implement the JMS specification.
- JMS message
  - an object that is used to communicate information between JMS clients (from producers to consumers).
  - Java objects can be serialised to JMS messages

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.

References

- Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, Section 5.1
- Dr J. Bacon, University of Cambridge