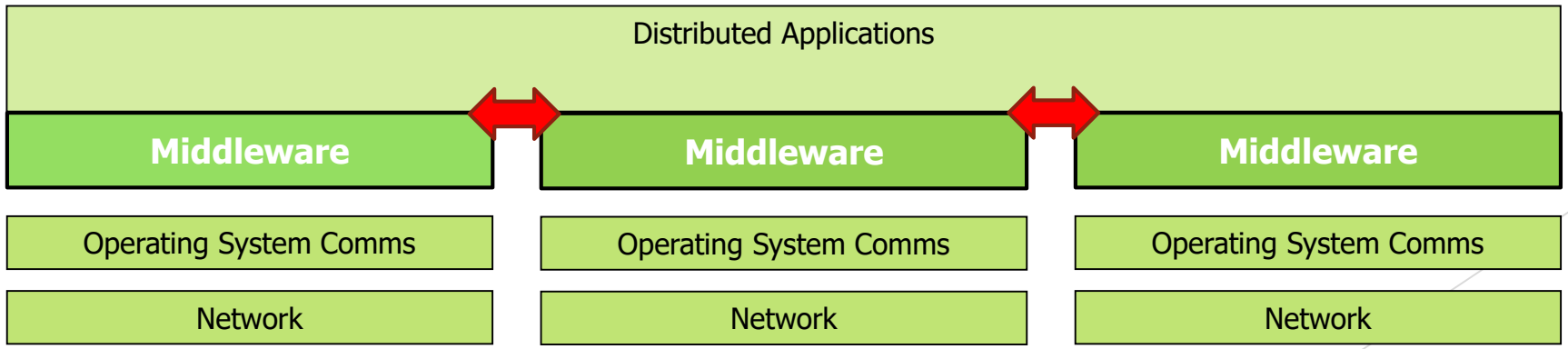
Middleware and Communication Patterns

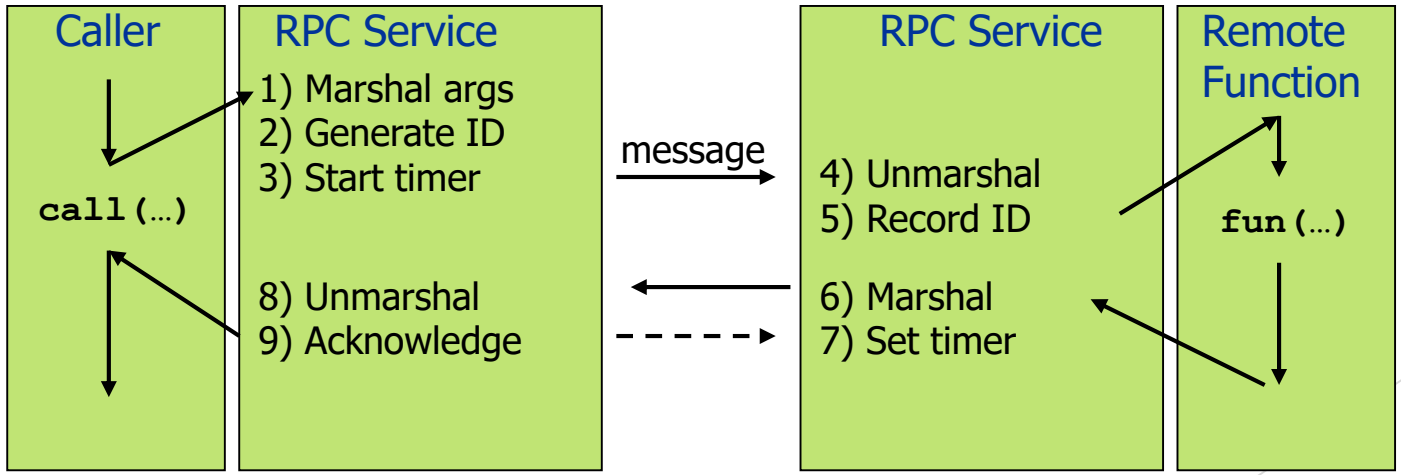


Classical Communication Patterns for middleware : a first characteristic

- They are :
  - Remote procedure call
  - Object oriented middleware
  - Message oriented middleware
  - Event based middleware and complex event processing

(1) Remote Procedure Call (RPC)

- Masks remote function calls as being local
- Client/server model
- Request/reply paradigm usually implemented with message passing in RPC service
- Marshalling of function parameters and return value

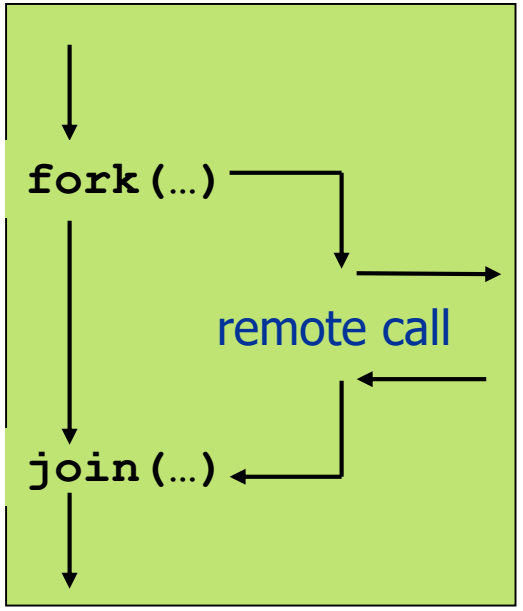


Properties of RPC

- 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 of requests and replies
- Distribution transparency (in the no-failure case)
  - hides the complexity of a distributed system

Disadvantages and limitations of RPC

- Synchronous request/reply interaction
  - tight coupling between client and server
  - client may block for a long time if server loaded
  - leads to multi-threaded programming in client
  - slow/failed clients may delay servers when replying
  - multi-threading essential for servers
- Distribution Transparency
  - Not possible to mask all problems
- RPC paradigm is not object-oriented
  - invoke functions on servers as opposed to methods on objects



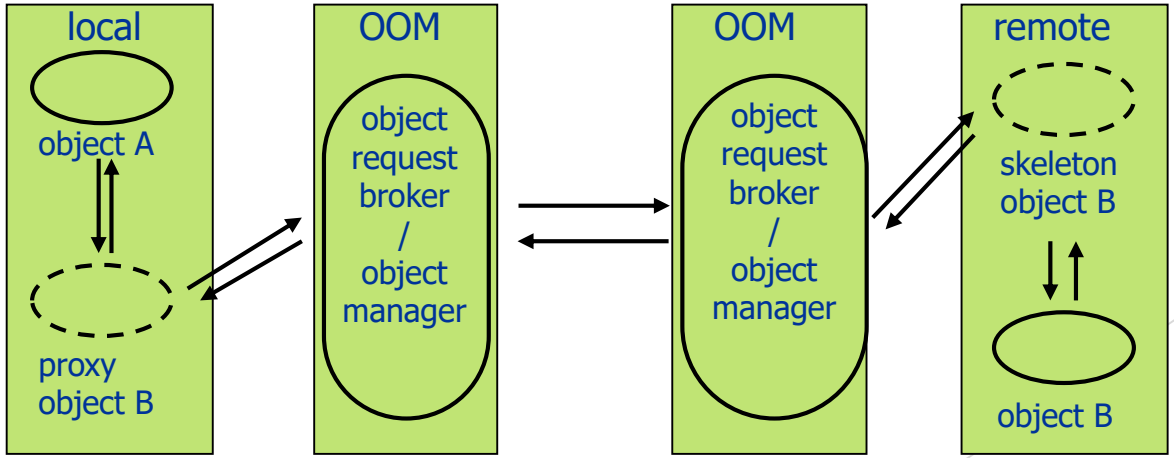
Do you know ?

- Any example for RPC based Middleware ?
- in your background …

- Example :
  - See XML-RPC : http://www.tutorialspoint.com/xml-rpc/
  - One kind of Web Service Middleware Communication paradigm is RPC
    - See W3C consortium : http://www.w3schools.com/webservices/

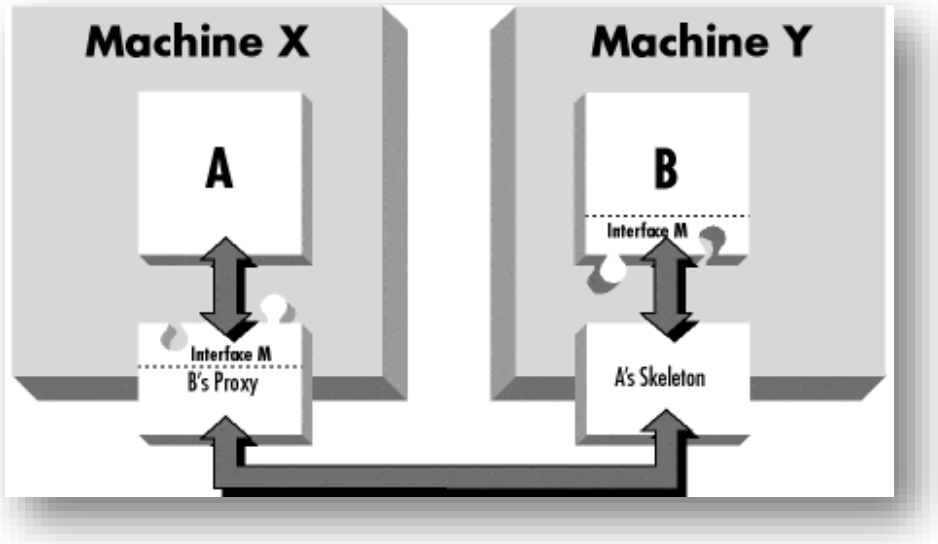
(2) Object-Oriented Middleware (OOM)

- 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 (were also in some RPC systems)
- Synchronous request/reply interaction
  - same as RPC
- Location Transparency
  - system (ORB) maps object references to locations



Do you know ?

- Any example for OOM ?
- in your background …
- Examples …

Java Remote Method Invocation (RMI)

- Covered in Java programming
- Distributed objects in Java

```
public interface PrintService extends Remote {
  int print(Vector printJob) throws RemoteException;
}
```

- RMI compiler creates proxies and skeletons
- RMI registry used for interface lookup
- Entire system written in Java (single-language system)

CORBA

- Common Object Request Broker Architecture
  - Open standard by the OMG (Version 3.0)
  - Language and platform independent

- **Object Request Broker** (ORB)
  - General Inter-ORB Protocol (GIOP) for communication
  - Interoperable Object References (IOR) contain object location
  - CORBA **Interface Definition Language** (IDL)
    - Stubs (proxies) and skeletons created by IDL compiler

CORBA IDL

- Definition of language-independent remote interfaces
  - Language mappings to C++, Java, Smalltalk, …
  - Translation by IDL compiler
- Type system
  - basic types: long (32 bit), long long (64 bit), short, float, char, boolean, octet, any, …
  - constructed types: struct, union, sequence, array, enum
  - objects (common super type Object)
- Parameter passing
  - in, out, inout
  - basic & constructed types passed by value
  - objects passed by reference

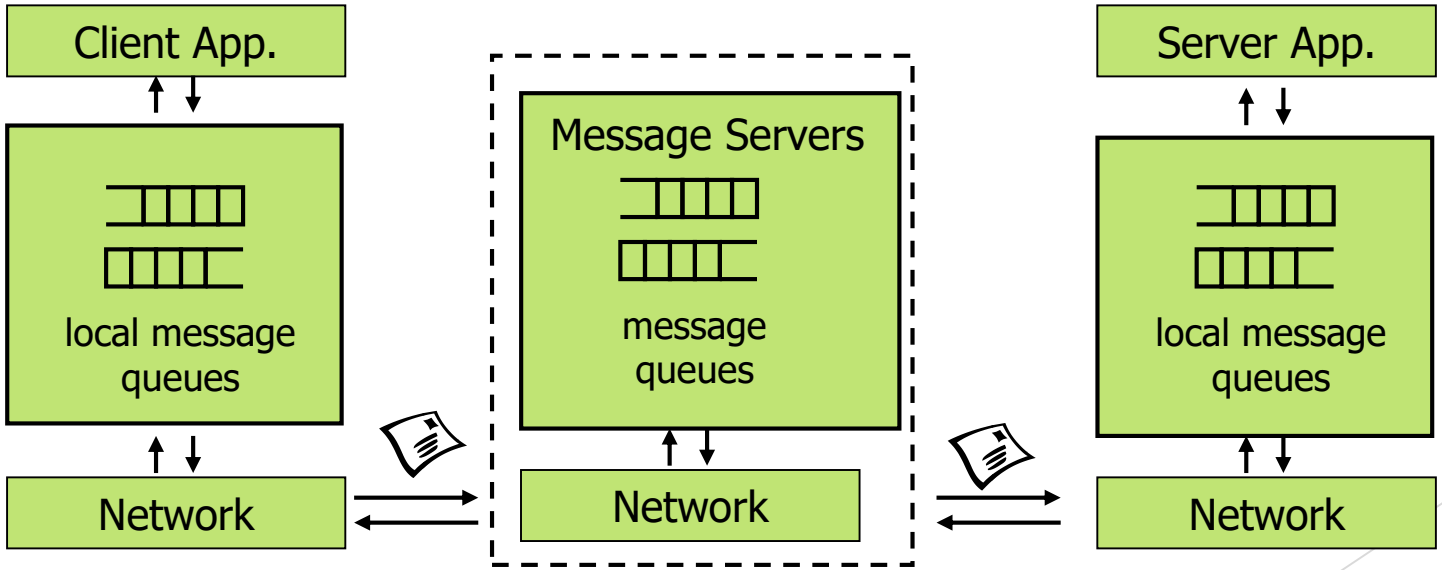
```
typedef sequence<string> Files;
interface PrintService : Server {
  void print(in Files printJob);
};
```

Advantages and Disadvantages of OOM

- Totally transparent distributed programming
- Synchronous request/reply interaction only
  - So CORBA oneway semantics added 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
  - Unadapted for ubiquitous systems and embedded devices

(3) Message-Oriented Middleware (MOM)

- Communication using messages
- Messages stored in message queues
- message servers decouple 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
- Processing of messages by intermediate message server(s)
  - May do filtering, transforming, logging, …
  - Networks of message servers

Java Message Service (JMS)

- API specification to access MOM implementations
- Two modes of operation *specified*:
  - Point-to-point
    - one-to-one communication using queues
  - Publish/Subscribe
    - cf. One pattern for Event-Based Middleware (ex . Java)
- JMS Server implements JMS API
- JMS Clients connect to JMS servers
- Java objects can be serialised to JMS messages

Disadvantages of MOM

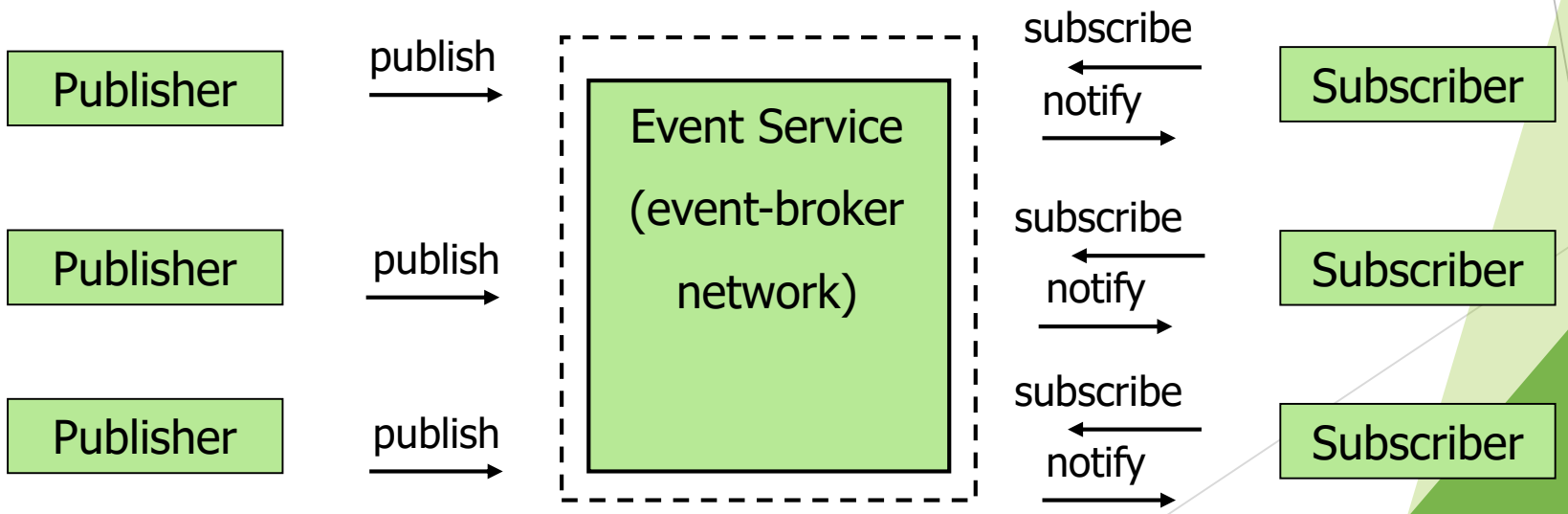
- Poor programming abstraction (but has evolved)
  - Rather low-level (cf. Packets)
  - Request/reply more difficult to achieve, but can be done
- Message formats originally unknown to middleware
  - No type checking (JMS addresses this – implementation?)
- Queue abstraction only gives one-to-one communication
  - Limits scalability (JMS pub/sub – heavy implementation of event based communications)

(4) Event-Based Middleware

- 1 emitter – N receiver
- With broadcast communications (ex. UDP)
- With unicast communications or peer to peer (ex. TCP), multiple communications are required

(4) Event-Based Middleware, ex. Publish/Subscribe Pattern

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

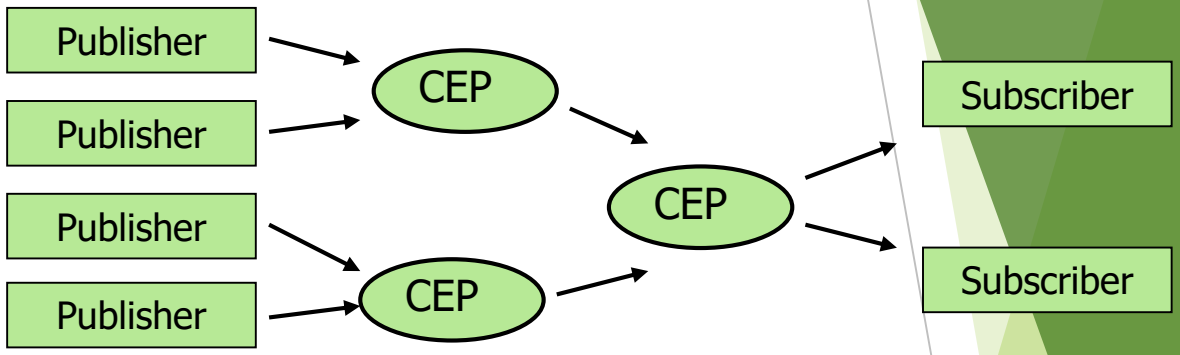


Properties of Publish/Subscribe

- Asynchronous communication
  - Publishers and subscribers are loosely coupled
- Many-to-many interaction between pubs. and subs.
  - Scalable scheme for large-scale systems
  - Publishers do not need to know subscribers, and vice-versa
  - Dynamic join and leave of pubs, subs
- (Topic and) Content-based pub/sub very expressive
  - Filtered information delivered only to interested parties

Complex event Processing (CEP)

- Composite Event Processing (CEP)
  - Events produce events after processing



- Example of CEP : Composite Event Detection (CED)
  - Content-based pub/sub may not be expressive enough
    - Potentially thousands of event types (primitive events)
    - Subscribers interest: event patterns
  - Composite Event Detectors (CED)
    - Subscribe to primitive events and publish composite events

- Alternative Implementation … (need multicast communications)

Summary

1. Remote Procedure Call
2. Object-Oriented Middleware
3. Message-Oriented Middleware
4. Event-Based Middleware

Example : Next MQTT Tutorial

- Middleware is an important abstraction for building distributed systems
- Synchronous vs. asynchronous communication
- Scalability, many-to-many communication
- Language integration
- Ubiquitous systems, mobile systems