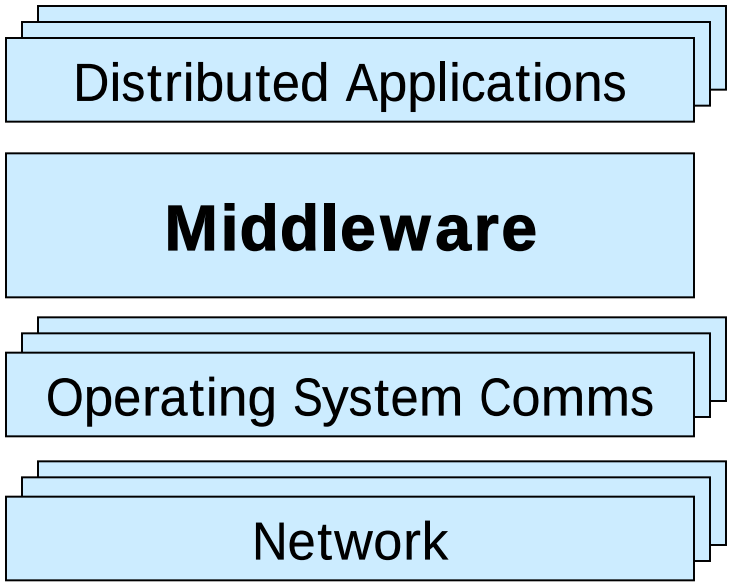
Middleware

Introduction to Middleware I

- What is Middleware?
  - Layer between OS and distributed applications
  - Hides complexity and heterogeneity of distributed system
  - Bridges gap between low-level OS communications and programming language abstractions
  - Provides common programming abstraction and infrastructure for distributed applications
  - Overview at: *http://www.middleware.org*



(remote calls, object invocation, messages, ...)

(sockets, IP, TCP, UDP, ...)

(packets, bits, ...)

Introduction to Middleware II

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

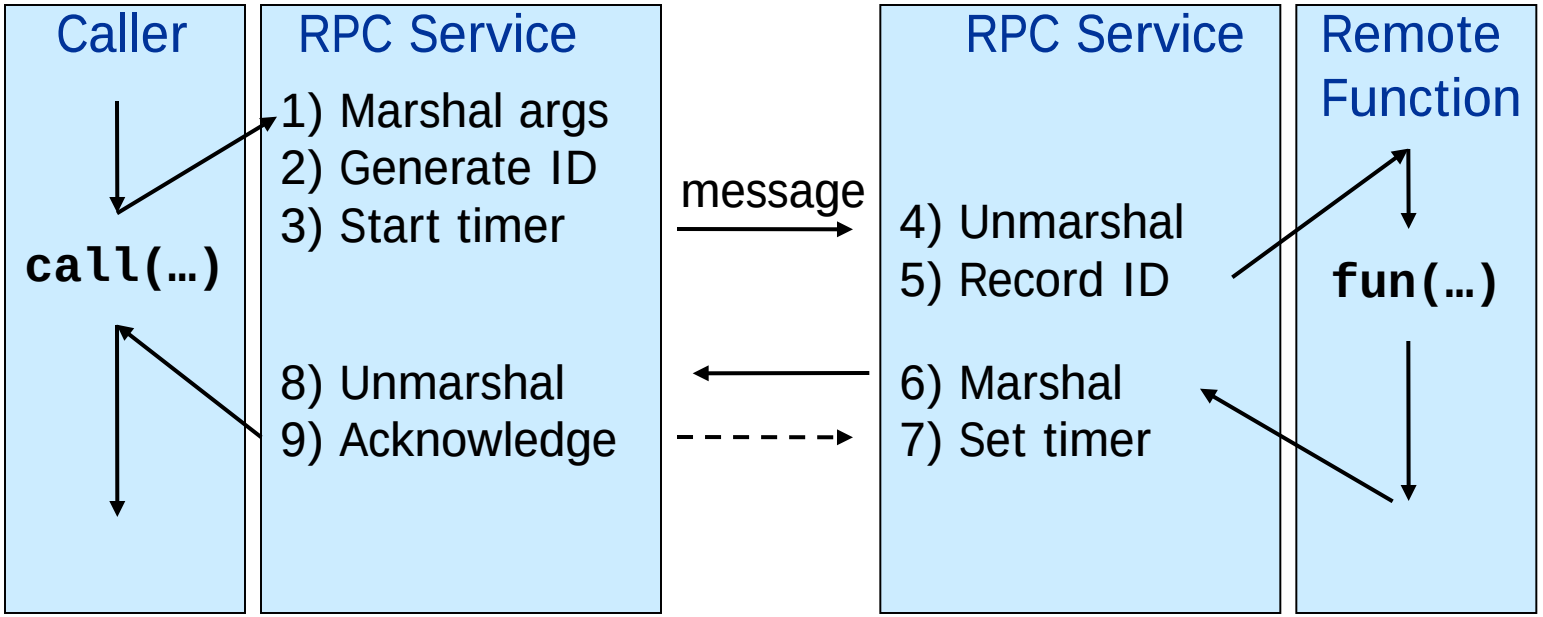
- Middleware dimensions:
  - Request/Reply vs. Asynchronous Messaging
  - Language-specific vs. Language-independent
  - Proprietary vs. Standards-based
  - Small-scale vs. Large-scale
  - Tightly-coupled vs. Loosely-coupled components

Outline

- Part I: Remote Procedure Call (RPC)
  - Historic interest, but still ubiquitous
- Part II: Object-Oriented Middleware (OOM)
  - Java RMI
  - CORBA
  - Reflective Middleware
- Part III: Message-Oriented Middleware (MOM)
  - Java Message Service
  - IBM MQSeries
  - Web Services
- Part IV: Event-Based Middleware
  - Cambridge Event Architecture
  - Hermes

Part I: 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 matching of requests and replies

Distribution transparency (in the no-failure case)

- hides the complexity of a distributed system

Various **reliability** guarantees

- deals with some distributed systems aspects of failure

Failure Modes of RPC

- Invocation semantics supported by RPC in the light of:
  - network and/or server congestion,
  - client, network and/or server failure
    - note DS independent failure modes
- RPC systems differ, many examples, local Cambridge thing was Mayflower

Maybe or at most once (RPC system tries once)

- Error return – programmer may retry

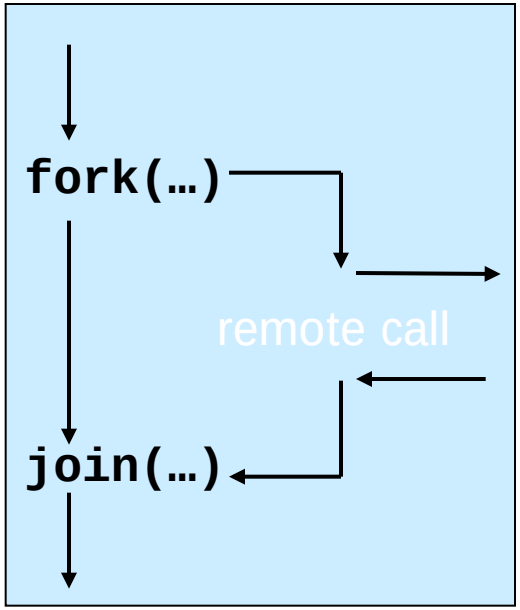
Exactly once (RPC system retries a few times)

- Hard error return – some failure most likely

  note that “exactly once” cannot be guaranteed

Disadvantages 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 at client
- slow/failed clients may delay servers when replying
  multi-threading essential at servers

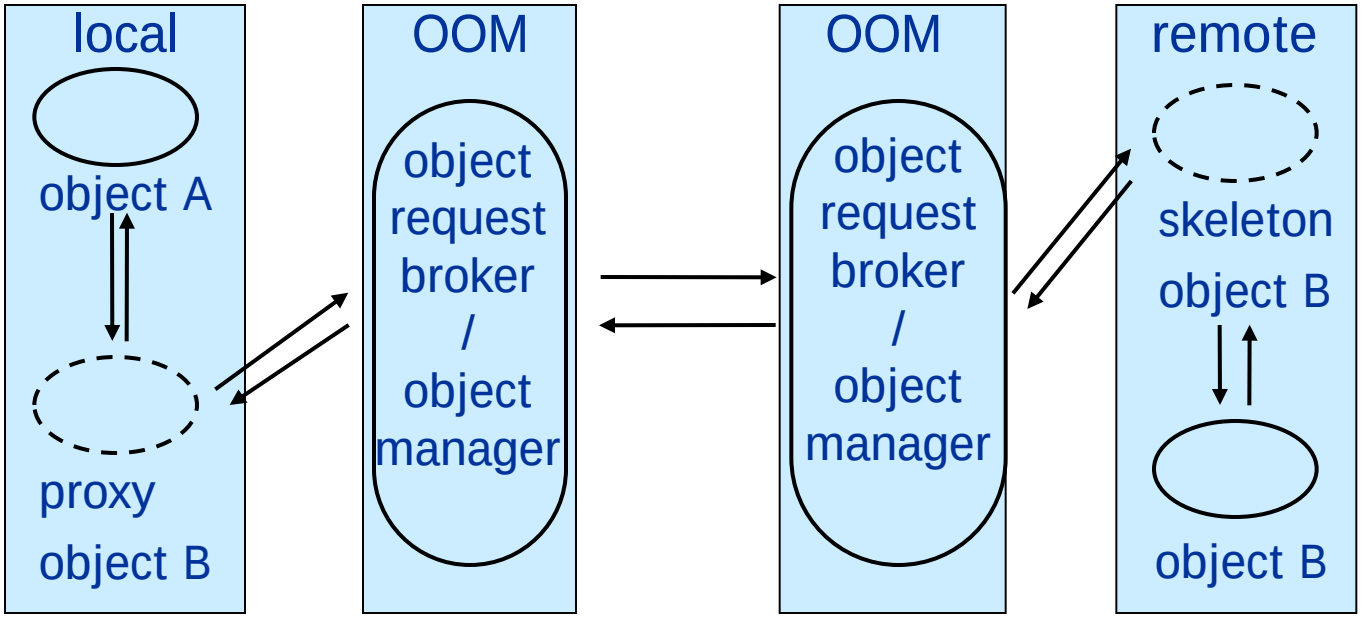


✖ Distribution Transparency
- Not possible to mask all problems

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

Part II: 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 e.g. Mayflower)

Synchronous request/reply interaction

- same as RPC

Location Transparency

- system (ORB) maps object references to locations

Services comprising multiple servers are easier to build with OOM

- RPC programming is in terms of ***server-interface (operation)***
- RPC system looks up server address in a location service

Java Remote Method Invocation (RMI)

- 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; other languages can be made to work with varying amounts of pain)

CORBA

- **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
  - 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
  - Dynamic remote method invocation
- **Interface Repository**
  - Querying existing remote interfaces
- **Implementation Repository**
  - Activating remote objects on demand

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`**)

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

- Parameter passing
  - **`in`**, **`out`**, **`inout`**
  - basic & constructed types passed by value
  - objects passed by reference

CORBA Services (selection)

- Naming Service
  - Names ➔ remote object references
- Trading Service
  - Attributes (properties) ➔ remote object references
- Persistent Object Service
  - Implementation of persistent CORBA objects
- Transaction Service
  - Making object invocation part of transactions
- Event Service and Notification Service
  - In response to applications‘ need for asynchronous communication
  - built above synchronous communication with *push* or *pull* options
  - *not* an integrated programming model with general IDL messages

Disadvantages of OOM

- ✖ Synchronous request/reply interaction only
  - So CORBA **oneway** semantics added and 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

OOM experience

Keynote address at Middleware 2009

Steve Vinoski

From Middleware Implementor to Middleware User

(There and back again)

Available from the course materials page

Reflective Middleware

- Flexible middleware (OOM) for mobile and context-aware applications – **adaptation** to context through *monitoring* and *substitution* of components

- Interfaces for **reflection**
  - Objects can inspect middleware behaviour

- Interfaces for **customisability**
  - Dynamic reconfiguration depending on environment
  - Different protocols, QoS, ...
  - e.g. use different marshalling strategy over unreliable wireless link

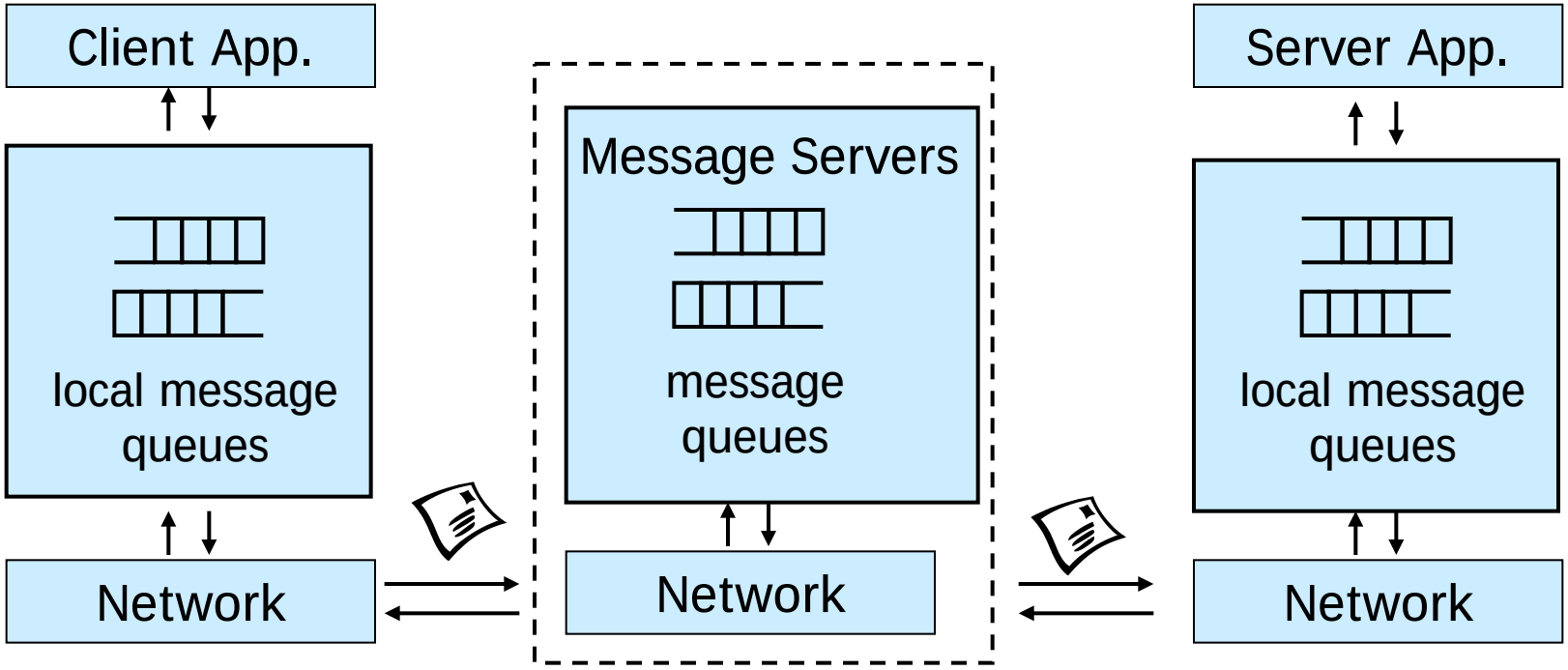
Part III: Message-Oriented Middleware (MOM)

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

Support for **reliable** delivery service

- Keep queues in persistent storage

Processing of messages by intermediate message server(s)

- May do filtering, transforming, logging, …
- Networks of message servers

Natural for database integration

IBM WebSphere MQ

- One-to-one reliable message passing using queues
  - Persistent and non-persistent messages
  - Message priorities, message notification
- **Queue Managers**
  - Responsible for queues
  - Transfer messages from input to output queues
  - Keep routing tables
- **Message Channels**
  - Reliable connections between queue managers
- Messaging API:

MQopen	Open a queue
MQclose	Close a queue
MQput	Put message into opened queue
MQget	Get message from local queue

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. Event-Based Middleware

- **JMS Server** implements JMS API
- JMS Clients connect to JMS servers
- Java objects can be serialised to JMS messages
- A JMS interface has been provided for MQ
- pub/sub (one-to-many) - just a specification?

Disadvantages of MOM

- ✖ Poor programming abstraction (but has evolved)
  - Rather low-level
  - Request/reply 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 – implementation?)

Web Services

- Use well-known web standards for distributed computing

Communication

- Message content expressed in **XML**
- **S**imple **O**bject **A**ccess **P**rotocol (**SOAP**)
  - Lightweight protocol for sync/async communication

Service Description

- **W**eb **S**ervices **D**escription **L**anguage (**WSDL**)
  - Interface description for web services

Service Discovery

- **U**niversal **D**escription **D**iscovery and **I**ntegration (**UDDI**)
  - Directory with web service description in WSDL

Properties of Web Services

Language-independent and open standard

SOAP offers OOM and MOM-style communication:

- Synchronous request/reply like OOM
- Asynchronous messaging like MOM
- Supports internet transports (http, smtp, ...)
- Uses XML Schema for marshalling types to/from programming language types

WSDL says how to use a web service

UDDI helps to find the right web service

- Exports SOAP API for access

Disadvantages of Web Services

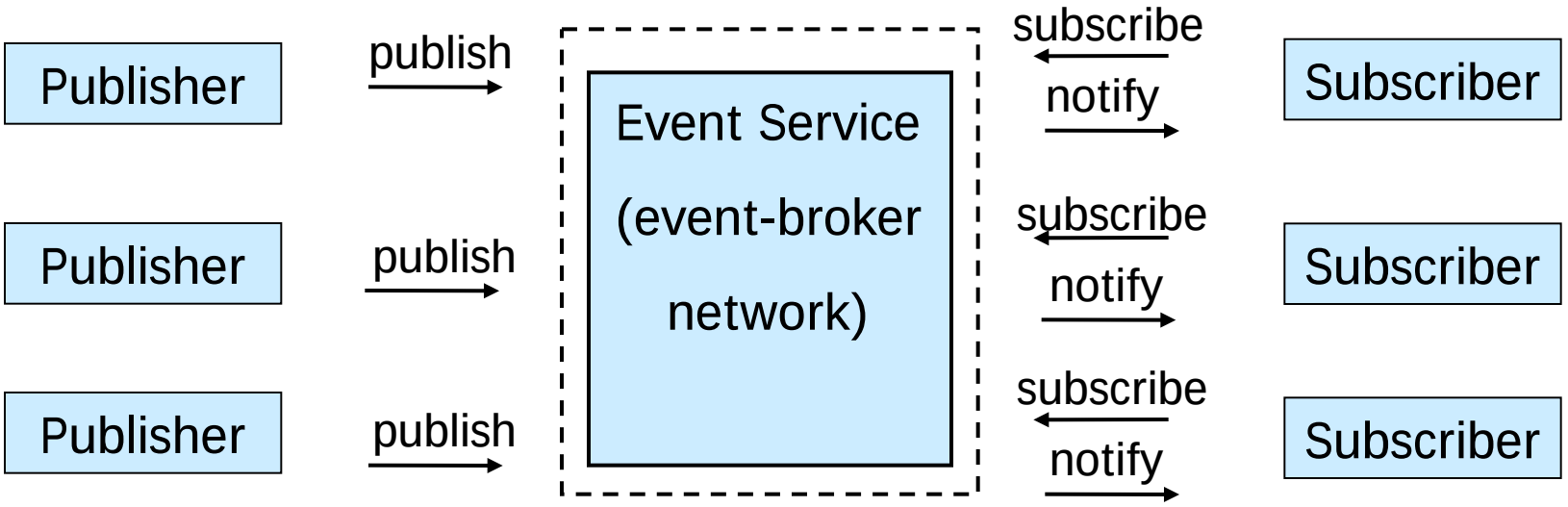
- ✖ Low-level abstraction
  - leaves a lot to be implemented
- ✖ Interaction patterns have to be built
  - one-to-one and request-reply provided
  - one-to-many?
  - still synchronous service invocation, rather than notification
  - no nested/grouped invocations, transactions, ...
- ✖ No location transparency

What we lack, so far

- ✖ General interaction patterns
  - we have one-to-one and request-reply
  - one-to-many? many to many?
  - notification?
  - dynamic joining and leaving?
- ✖ Location transparency
  - anonymity of communicating entities
- ✖ Support for pervasive computing
  - data values from sensors
  - lightweight software

Part IV: Event-Based Middleware a.k.a. Publish/Subscribe

- **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) or name/value pairs



Topic-Based and Content-Based Pub/Sub

- Event Service matches events against subscriptions
- What do subscriptions look like?

Topic-Based Publish/Subscribe

- Publishers publish events belonging to a **topic** or **subject**
- Subscribers subscribe to a **topic**

```
subscribe(PrintJobFinishedTopic, …)
```

(Topic and) Content-Based Publish/Subscribe

- Publishers publish events belonging to **topic**s and
- Subscribers provide a **filter** based on ***content*** of events

```
subscribe(type=printjobfinished, printer='aspen', …)
```

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, (brokers - see lecture DS-8)

(Topic and) Content-based pub/sub very expressive

- Filtered information delivered only to interested parties
- Efficient content-based routing through a broker network

Composite Event Detection (CED)

Content-based pub/sub may not be expressive enough

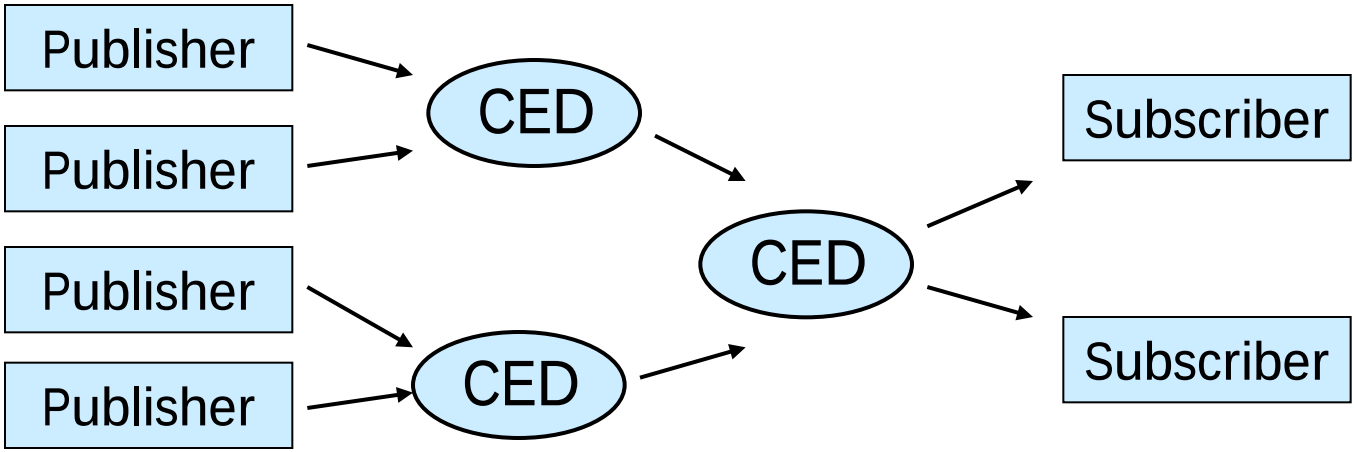
- Potentially thousands of event types (primitive events)
- Subscribers interest: event *patterns* (define *high-level* events, ref DS-2)

Event Patterns

- `PrinterOutOfPaperEvent` or `PrinterOutOfTonerEvent`

Composite **E**vent **D**etectors (CED)

- Subscribe to primitive events and publish composite events



Summary

- Middleware is an important abstraction for building distributed systems

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

- Synchronous vs. asynchronous communication
- Scalability, many-to-many communication
- Language integration
- Ubiquitous systems, mobile systems