CMPU4021 Distributed Systems

Revision 2

RMI - Introduction

- Communication among distributed objects via RMI
  - Recipients of remote invocations are remote objects, which implement remote interfaces for communication
- Reliability
  - Either one or both the invoker and invoked can fail, and status of communication is supported by the interface (e.g., notification on failures, reply generation, parameter processing – marshalling/unmarshalling)
    - *Marshalling* – the process of taking a collection of data items and assembling them into a form suitable transmission in a message (external data representation)
    - *Unmarshalling* – disassembling them on arrival to produce an equivalent collection of data items at the destination.
- Local invocations target local objects, and remote invocations target remote objects

Interfaces

- Interfaces hide the details of modules providing the service; and access to module variables is only indirectly via ‘getter’ and ‘setter’ methods / mechanisms associated with the interfaces
  - e.g., call by value/reference for local calls through pointers vs. input, output, and input paradigms in RMI through message-data and objects

- *Service interfaces*
  - client-server model, specification of the procedures offered by a server
    - defining the types of input and output arguments

- *Remote interfaces*
  - distributed object model, specifies the methods of an object that are available for invocation by objects in other processes
    - defining the types of the input and output arguments of each of them.

Interface definition languages (IDLs)

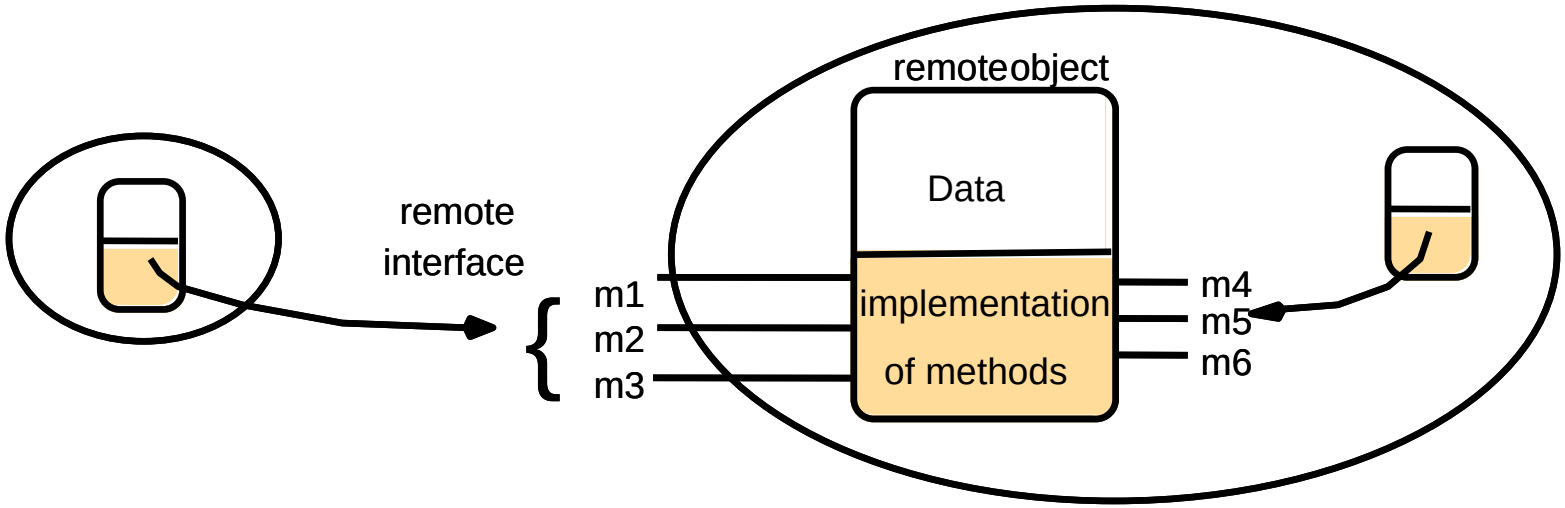
IDL

- Provides a ‘generic’ template of interfaces for objects in different languages to perform remote invocation among each other
  - E.g., CORBA IDL, Java RMI

- Provides a notation for defining interfaces in which each of the parameters of a method may be described as for *input* or output in addition to having its type specified.

The distributed object model:Remote interfaces

- Remote objects have a class that implement remote methods (as public).
- In Java, a remote interface class extends the `Remote` interface
- Local objects can access methods in an interface plus methods implemented by remote objects (Remote interfaces can't be constructed – no constructors)



Invocation Semantics

- In local OO system
  - all methods are invoked exactly once per request –guaranteed –unless whole process fails
- In distributed object system, we need to know what has happened if we do not hear result from remote object i.e. did the request go missing, did the response go missing
- 3 different types of guarantee (invocation semantics) may be provided
  - could be implemented in a middleware platform intended to support remote method invocations:
    - *Maybe*
    - *At-Least-Once*
    - *At-Most-Once*

Maybe Invocation Semantics

- If the invoker cannot tell whether a remote method has been invoked or not

- Very inexpensive, but only useful if the system can tolerate occasional failed invocations

At-Least-Once Invocation Semantics

- If the invoker receives a result, then it is guaranteed that the method was invoked at least once
- Achieved by resending requests to mask omission failure
- Only useful if the operations are idempotent (x = 10, rather than x = x + 10)
- Inexpensive on server

At-Most-Once Invocation Semantics

- If the invoker receives a result, then it is guaranteed that the method was invoked only once
- If no result is received, then the method was executed either never or once
- Achieved by resending requests, and storing and resending responses
- More expensive on a server / remote object, which must maintain results and recognise duplicate messages

Invocation semantics: failure model

- Maybe, At-least-once and At-most-once
  - can suffer from crash failures when the server containing the remote object fails.

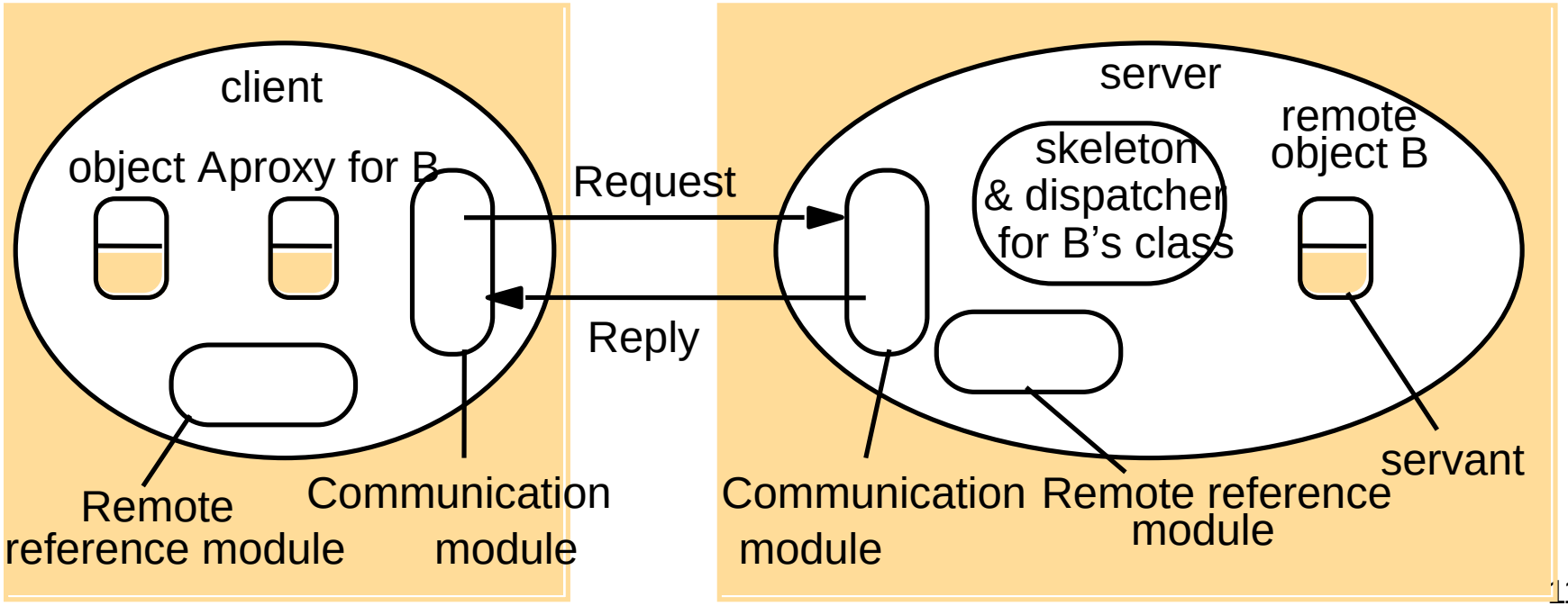
- *Maybe*
  - if no reply, the client does not know if method was executed or not
    - omission failures - if the invocation or result message is lost

- *At-least-once*
  - the client gets a result (and the method was executed at least once) or an exception (no result)
    - arbitrary failures. If the invocation message is retransmitted, the remote object may execute the method more than once, possibly causing wrong values to be stored or returned.
    - if *idempotent* operations are used, arbitrary failures will not occur

- At-most-once
  - the client gets a result (and the method was executed exactly once) or an exception (instead of a result, in which case, the method was executed once or not at all)

Implementation of RMI

- Several separate object and modules
- An application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference.



Java distributed garbage collection algorithm

Based on reference counting. Works with the local garbage collector as follows:

1. Each server process maintains a set of the names of the processes that hold remote object references for each of its remote objects. E.g. *B.holders* is a list of client processes referencing a remote object B.

2. When the client first receives a remote reference for B it makes an *addRef(B)* invocation on the server. The server adds C to *B.holders*.

3. When C's garbage collector notices that the proxy for B is no longer reachable it makes a *removeRef(B)* invocation on the server and deletes the proxy. The server deletes C from *B.holders.*

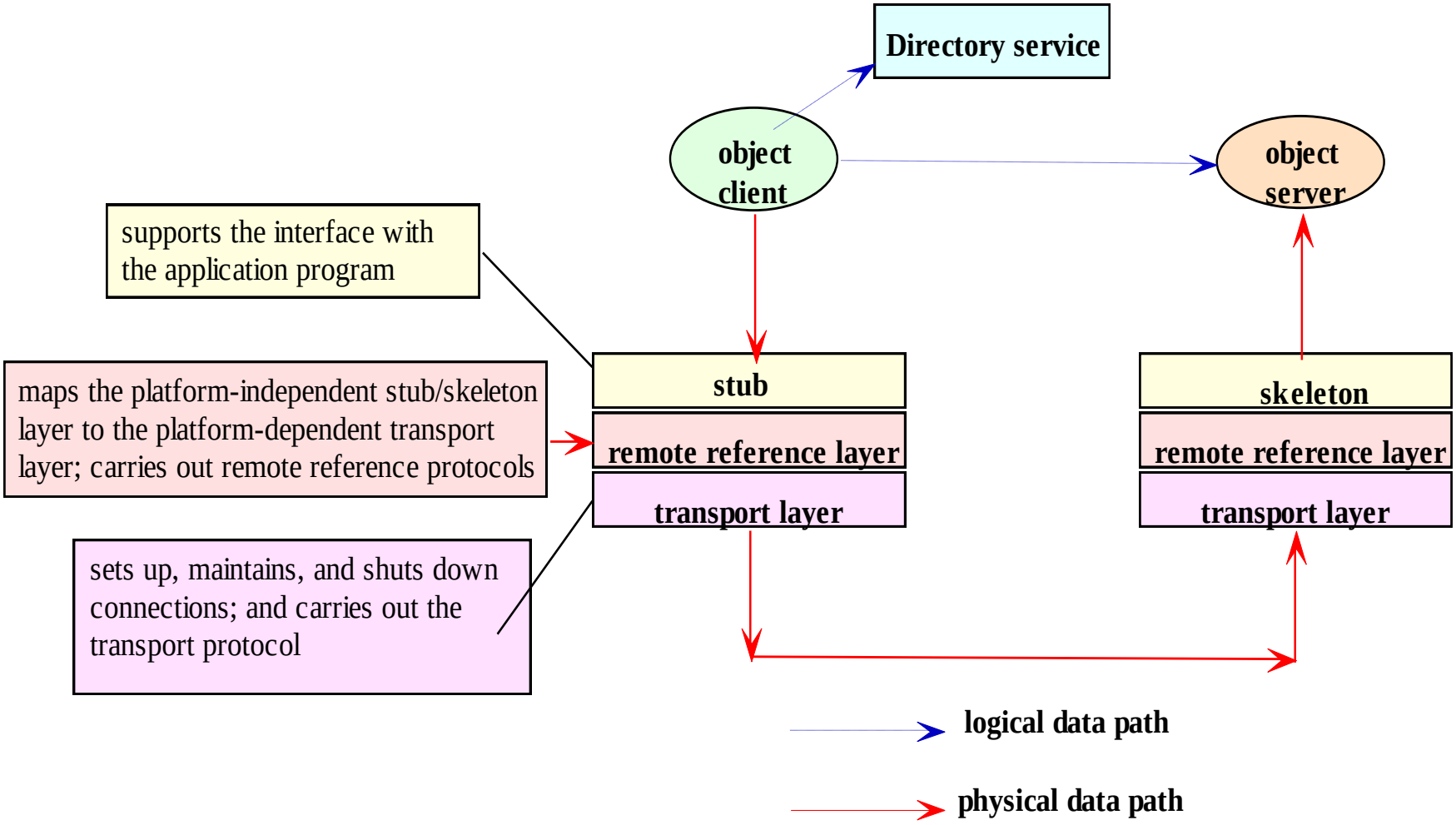
4. When *B.holders* is empty, the server's local garbage collector will reclaim the space occupied by B unless there are any local holders.

Java distributed garbage collection algorithm

- Failure handling:
  - Needs to be time delay between removal of last reference and garbage collection; duplication of messages, missing messages
  - *addRef* and *removeRef* are idempotent: *if an addRef(B)* call returns an exception, the client will not create a proxy but will make a *removeRef(B)* call. The effect of *removeRef* is correct whether or not the *addRef* succeeded.

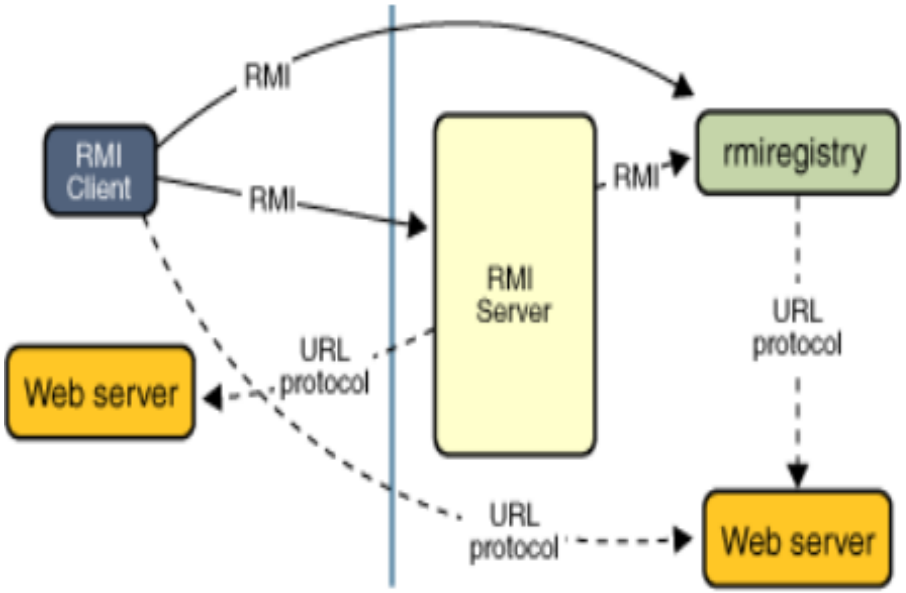
- Failures of client processes:
  - Servers *lease* their objects to clients for a limited period of time (starts with *addRef* and ends when time period elapses or when the client calls *removeRef).*

The Java RMI Architecture



RMI Distributed Application

- Uses the RMI registry to obtain a reference to a remote object.
- The server calls the registry to associate (or bind) a name with a remote object.
- The client looks up the remote object by its name in the server's registry and then invokes a method on it.
- The RMI system uses an existing web server to load class definitions
  - from server to client and from client to server, for objects when needed.



RMI Dynamic class loading

- Facilitates updates of remote objects once the application has been deployed.
- If the server implementation is changed and a new stub class is generated the stub class will have to be distributed to all clients.
  - This is undesirable.
- A better approach would be to make the new stub class available online and whenever a client starts up it automatically loads the new class from the web server.

RMI Dynamic Class Loading

- Both client and server need to install a security manager to be able to load classes remotely:

- Client and server need a `security policy` file granting the necessary rights like opening network connections (the following security policy file simply grants everything = no security at all):

`mysecurity.policy` file:

```
grant {
permission java.security.AllPermission;
}
```

- Starting the server with the security policy file, e.g.

```
java –Djava.rmi.server.codebase="http://myserver/example/"
–Djava.security.policy=mysecurity.policy MyServer
```

- Starting the client with the security policy file:

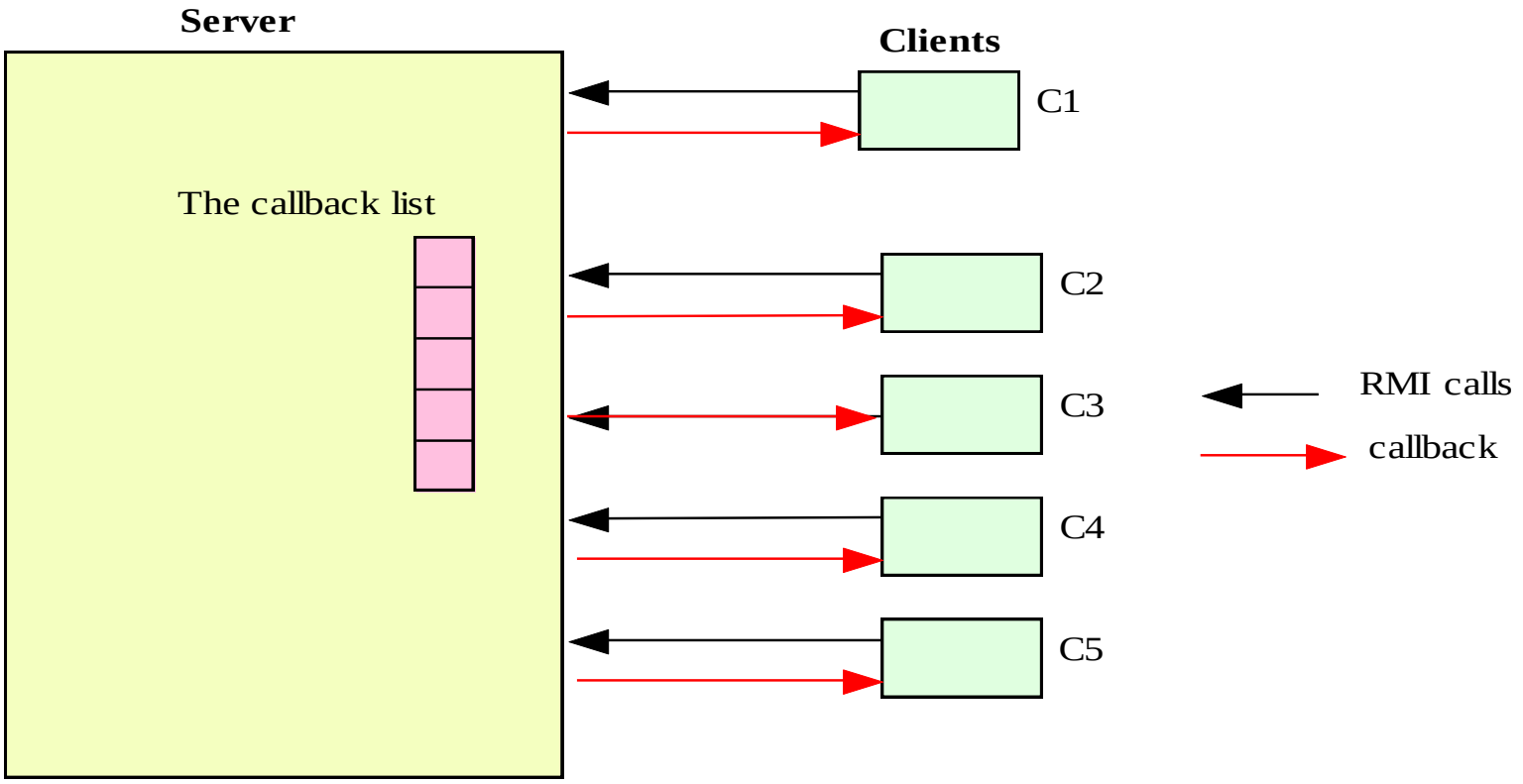
```
java –Djava.security.policy=mysecurity.policy MyClient
```

RMI Callbacks

- Like any other callback
- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.

RMI Callbacks

- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.



TRANSACTIONS

Introduction to transactions

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

- **A**tomicity:
  - *All or nothing*
  - The transaction happens as a single indivisible action. Everything succeeds or else the entire transaction is rolled back. Others do not see intermediate results.

- **C**onsistency:
  - 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

- **I**solated (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.

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

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'
  - 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.

Strict two phase locking

- Locks are only released upon commit / abort
- Extension of two-phase locking that prevents ***dirty reads*** and ***premature writes***

Rules for Strict Two-Phase Locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

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 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

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.

Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

This is a request with a reply

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

These are asynchronous requests to avoid delays

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

Asynchronous request

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

- participant interface- *canCommit?, doCommit, doAbort*
- coordinator interface- *haveCommitted, getDecision*

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.

The Voting Rules

1. Each participant has one vote which can be either 'commit‘ or 'abort’;

2. Having voted, a participant cannot change its vote;

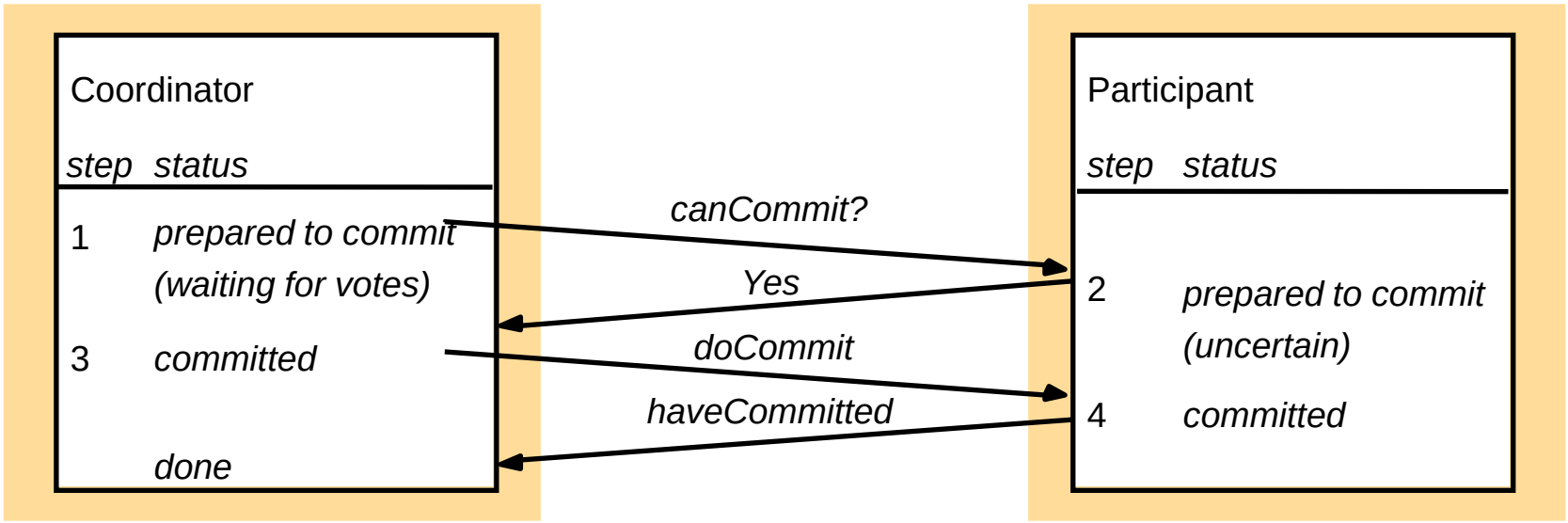
3. If a participant votes 'abort' then it is free to abort the transaction immediately; any site is in fact free to abort a transaction at any time up until it records a 'commit' vote. Such a transaction abort is known as a unilateral abort.

4. If a participant votes ‘commit’, then it must wait for the co-ordinator to broadcast either the 'global-commit' or 'global-abort' message;

5. If all participants vote 'commit' then the global decision by the co-ordinator must be 'commit';

6. The global decision must be adopted by all participants.

Communication in two-phase commit protocol



- Time-out actions in the 2PC
  - to avoid blocking forever when a process crashes or a message is lost
  - *uncertain* participant (step 2) has voted yes. it can't decide on its own
    - it uses `getDecision` method to ask coordinator about outcome
  - participant has carried out client requests, but has not had a `Commit?` from the coordinator. It can abort unilaterally
  - coordinator delayed in waiting for votes (step 1). It can abort and send `doAbort` to participants.

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 PROTOCOL

Three-phase commit (3PC) protocol

Phase 1 (the same as for two-phase commit):

- The coordinator sends a `canCommit?` request to each of the participants in the transaction.
- 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:

- 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.

Phase 3:

- 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`.

WEB SERVICES

Web Services

- Set of protocols by which services can be published, discovered, and used in a technology neutral form
  - Language & architecture independent

- Provide a basis where a client program in one organisation may interact with a server in another organisation without human supervision.

- Based on the ability to use an HTTP request to cause the execution of a program.
  - An result is produced by called program and then returned.

Web Services

- General principles
  - Payloads are text (XML or JSON)
    - Technology-neutral
  - HTTP used for transport
    - Use existing infrastructure: web servers, firewalls, load-balancers

- Web server
  - Provides a basic HTPP service

- Web service
  - Provides a service based on the operation defined in its interface.

- Applications will typically invoke multiple remote services
  - Service Oriented Architecture (SOA)
  - SOA = Programming model

Web services infrastructure and components

- Web services and applications may be built on top other web services.
- Some particular web services provide general functionality required for the operation of a large number of other web services:
  - Directory services,
  - Security
- A web service generally provides a *service description*, which includes an interface definition and other information, such as the server's URL

Web Services vs. Distributed Objects

Web Services

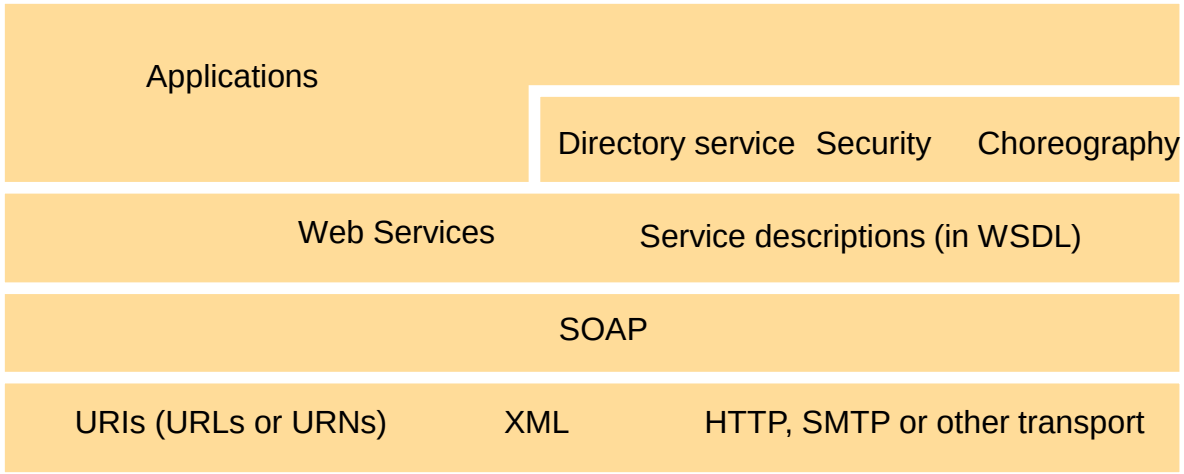
- Document Oriented
  - Exchange documents
- Document design is the key
  - Interfaces are just a way to pass documents
- Stateless computing
  - State is contained within the documents that are exchanged (e.g., customer ID)

Distributed Objects

- Object Oriented
  - Instantiate remote objects
  - Request operations on a remote object
  - Receive result
  - …
  - Eventually release the object
- Interface design is the key
  - Data structures just package data
- Stateful computing
  - Remote object maintains state

Web services infrastructure and components

- Typical communication architecture in which web services operate:
  - A web service is identified by URI and can be accessed by clients using messages formatted in XML.
  - Simple Object Access Protocol (SOAP) is used to encapsulate these messages and transmit them over HTTP or another protocol, e.g. TCP or SMTP.
  - A web service deploys service descriptions to specify the interface and other aspects of the service for the benefit of potential clients



URI, URL and URN

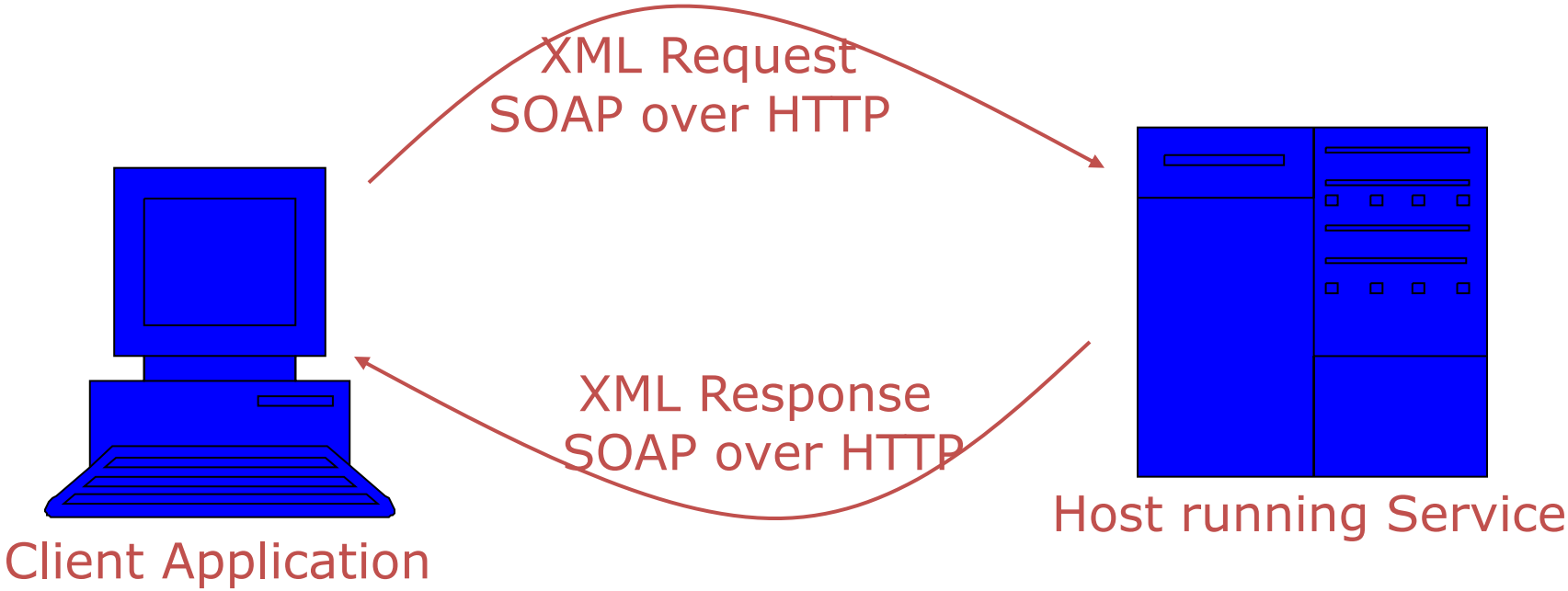
- The Uniform Resource Identifier (URI)
  - a general resource identifier, whose value may be either a URL or a URN.
    - Uniform Resource Names (URNs) are location independent
      - they rely on a lookup service to map them onto the URLs of resources.

SOAP

- Designed to enable both client-server and asynchronous interaction over the Internet.
- It defines a scheme for using XML to represent
  - the contents of request and reply messages
  - a scheme for the communication of documents.
- Objects marshaled and unmarshaled to SOAP-format XML
- SOAP is a messaging format
  - No garbage collection or object references
  - Does not define transport

Web Services

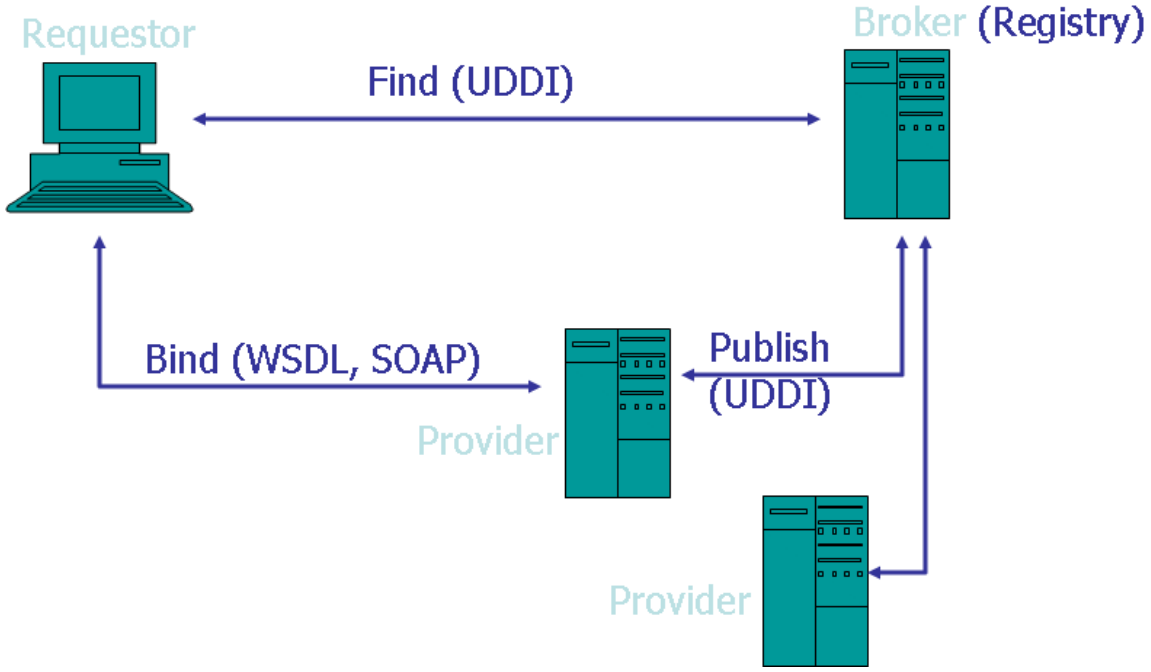
- Services offered via the web



Web Services Architecture

Architecture needs to cater for:

- Providing a list of available services
  - E.g.what credit checking web services are available for a mortgage service to use?)
- Providing a description or specification for the service
  - (e.g.… what parameters does the selected credit check service expect.. client name/PPS number?, what does it return? …approval rating?)
- Sending messages between the client (I.e. *requestor* of the service) and service…
  - (… message to credit rating service from client application, containing client name etc.?)



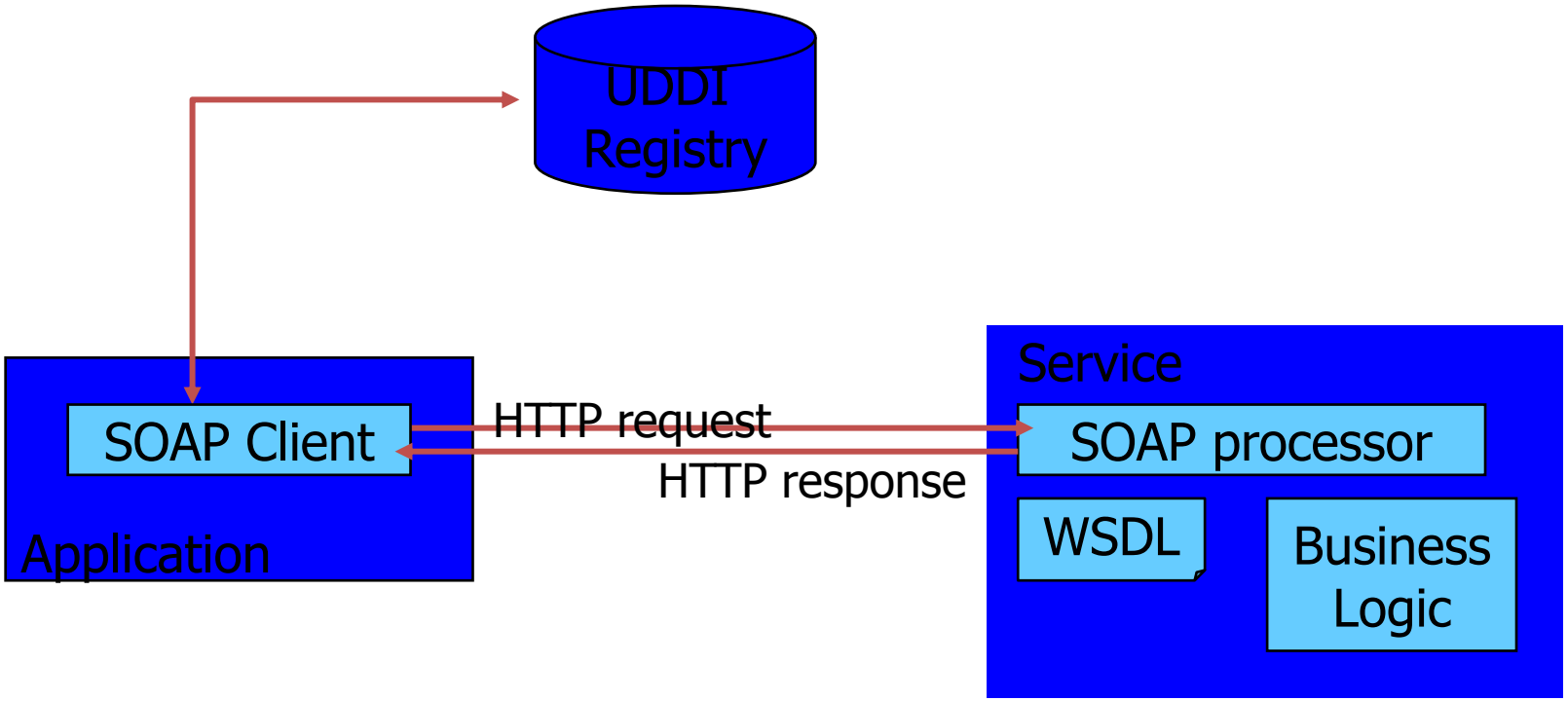
Technologies

Many implementations/frameworks for web services but common technologies:

- **SOAP** (Simple Object Access Protocol)
  - provides a way to communicate between applications running on different operating systems, with different technologies and programming languages
  - XML based protocol (on HTTP)
- **WSDL** (Web Service Description Language)
  - machine-readable descriptions of Web services interfaces.
  - XML based
- **UDDI (**Universal Description, Discovery and Integration)
  - UDDI provides an interface for publishing and updating information about web services.

Technologies in the Process

1. client queries UDDI registry for a service
   – by name, category, identifier or some other criteria stored by registry
2. client then obtains information about location of WSDL doc from UDDI registry
3. WSDL doc contains info about how to contact service and format of request msg
4. client creates SOAP msg in accordance with WSDL and sends request to host where service is
5. service responds with a SOAP msg to indicate results of service request



SOAP

- Simple Object Access Protocol
  - Extension of XML-RPC

- A SOAP message is an ordinary XML document containing the following elements:
  - An Envelope element that identifies the XML document as a SOAP message
  - A Header element that contains header information
  - A Body element that contains call and response information
  - A Fault element containing errors and status information

- Scheme for
  - Using XML to represent the contents of request and reply messages
  - Communication of documents

- Originally transported only over HTTP POST
  - Can also be transported over SMTP (e-mail), TCP, UDP

Service Descriptions

- Interface definitions needed to allow clients to communicate with services
- Java Remote Interfaces ~= IDL ~= WSDL
- WSDL (Web Services Description Language)
  - Describe operations
  - Provide URI
  - Identify Transport Protocol
- All the information required by the client
- Describes either
  - Types of messages it can receive
  - Types of operations it can perform

Directory Service

- A way to obtain service descriptions.
- UDDI (Universal Description, Discovery and Integration) of Web Services
  - An industry effort to define a searchable registry of services
- UDDI provides provides both
  - a name service and
  - a directory service

How is UDDI used?

- By business analysts to search for services
  - similar to search engine
  - portals needed

- By developers to publish services and to write software to use the discovered services

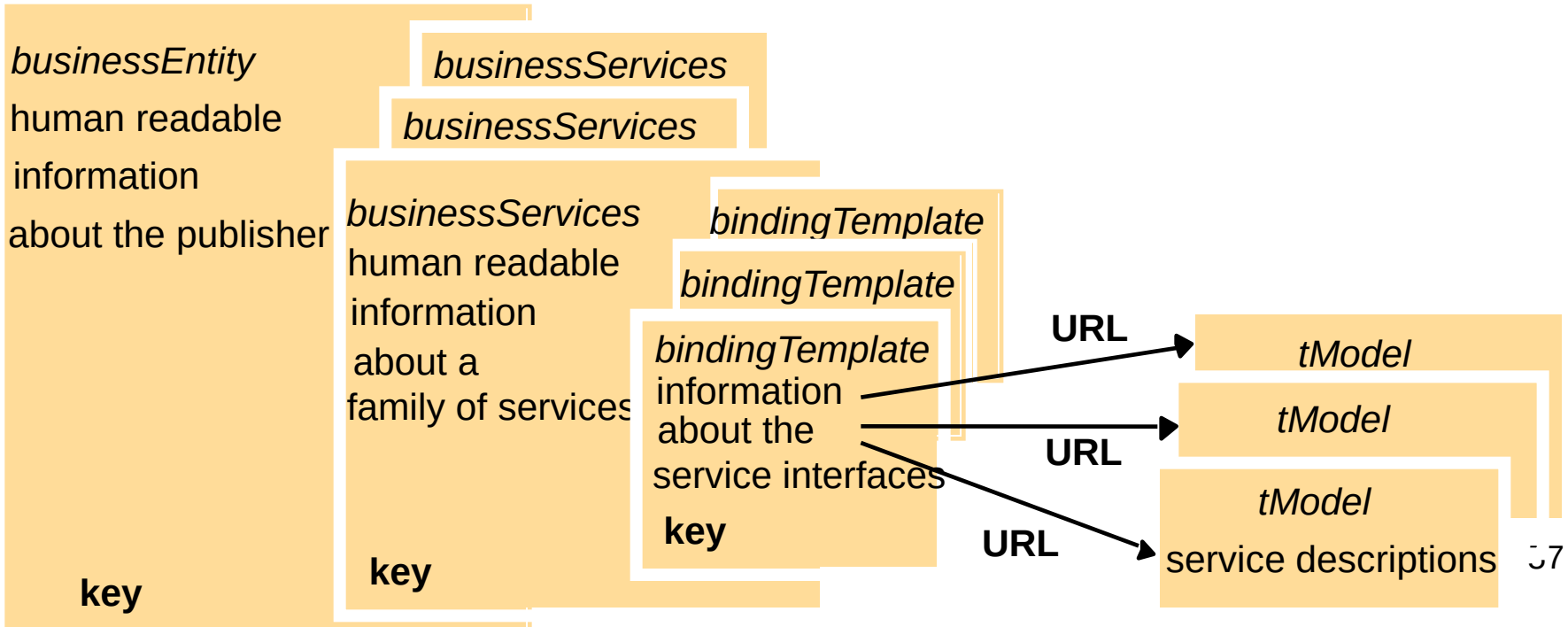
- Incorporated into toolkits to automate publishing of services

- Clients may use the yellow pages approach to look up a particular category of service, such as travel agent or bookseller, or they may use the white pages approach to look up a service with reference to the organization that provides it.

- The data structures supporting UDDI are designed to allow all the above styles of access and can incorporate any amount of human-readable information.

UDDI: Data Structures

- *businessEntity*
  - describes the organization that provides these web services, giving its name, address and activities, etc.;
- *businessServices*
  - stores information about a set of instances of a web service, such as its name and a description of its purpose (for example, travel agent or bookseller);
- *bindingTemplate*
  - holds the address of a web service instance and references to service descriptions;
- *tModel*
  - holds service descriptions, usually WSDL documents, stored outside the database and accessed by means of URLs.



Comparison of web services with distributed object model

- A web service has a service interface which can provide operations for accessing and updating the data resource it manages.

- At a superficial level, the interaction between client and server is similar to RMI, where a client uses a remote object reference to invoke an operation in a remote object.
  - For a web service, the client uses a URI to invoke an operation in the resource named by that URI.

Comparison of web services with distributed object model

- Remote object references are not very similar to URIs
  - The URI of a web service can be compared with the remote object reference of a single object.
  - However, in the distributed object model, objects can create remote objects dynamically and return remote reference to them.
  - The recipient of these remote references can use them to invoke operations in the object to which they refer.

Comparison of web services with distributed object model

- Servants
  - In the distributed object model, the server program is generally modelled as a collection of servants (potentially) remote objects.
  - In contrast, web services do not support servants. Therefore, web services applications cannot create servants as and when they are needed to handle different resources.
    - To enforce this situation, the implementation of web service interfaces must not have either constructors or main methods.

REST

Representational State Transfer (REST)

- The key characteristic of most web services is that they can process XML formatted SOAP messages
  - An alternative is the REST approach

- REST is a web standards based architecture
  - Uses HTTP Protocol for data communication
  - Resource-oriented
    - every component is a resource
    - a resource is accessed by a common interface using HTTP standard methods

- Clients use URLs and the HTTP operations GET, PUT, DELETE and POST to manipulate resources
  - The emphasis is on the manipulation of *data resources* rather than on interfaces.

REST

- When a new resource is created, it has a new URL by which it can be accessed or updated.
  - Clients are supplied with the entire state of a resource instead of calling an operation to get some part of it.

- The Amazon web services may be accessed either by SOAP or by REST

RESTful Web Services

- A web service is:
  - A collection of open protocols
  - Standards used for exchanging data between applications or systems
  - Interoperability between different languages (Java and Python) o platforms (Windows and Linux)

- Web services based on REST Architecture are known as RESTful Web Services
  - Use HTTP methods to implement the concept of REST architecture
  - URI (Uniform Resource Identifier) to define a RESTful service
  - Resources representation: JSON

REST

- Everything is a resource
- Any interaction of a RESTful API is an interaction with a resource.
- Resources are sources of information,
  - typically documents or services, or
  - Users (e.g. as a URL of their GitHub)

HTTP Methods in a REST REST based architecture

- GET
  - Provides a read only access to a resource.
- PUT
  - Used to create a new resource.
- DELETE
  - Used to remove a resource.
- POST
  - Used to update an existing resource or create a new resource.
- OPTIONS – Used to get the supported operations on a resource.

Coordination of web services

- The SOAP infrastructure supports single request-response interactions between clients and web services.
- Many applications involve several requests that need to be done in a particular order. The need for web services as clients to be provided with a description of a particular protocol to follow when interacting with other web services.
- For composite web services, a transactions management protocol such as 2PC is required
  - WS-Coordination
- Simpler approach is Web Service choreography
  - Global description of a set of interactions
  - Defines coordination
  - Enhances interaction

MIDDLEWARE

Middleware: Attributes

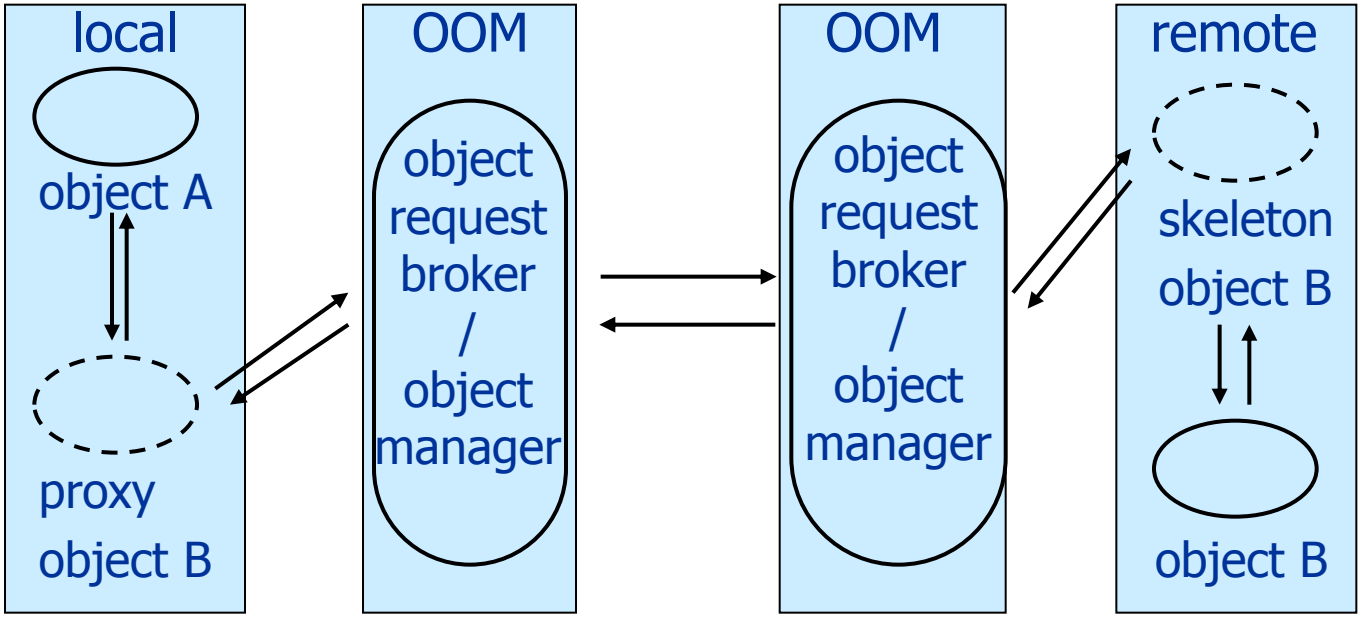
- Provides services to applications
- Requires system resources, dependencies
- Has vulnerabilities and constraints
- May or may not implement its own access control model
- Developer may not have control over its design

Common Middleware Services

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

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

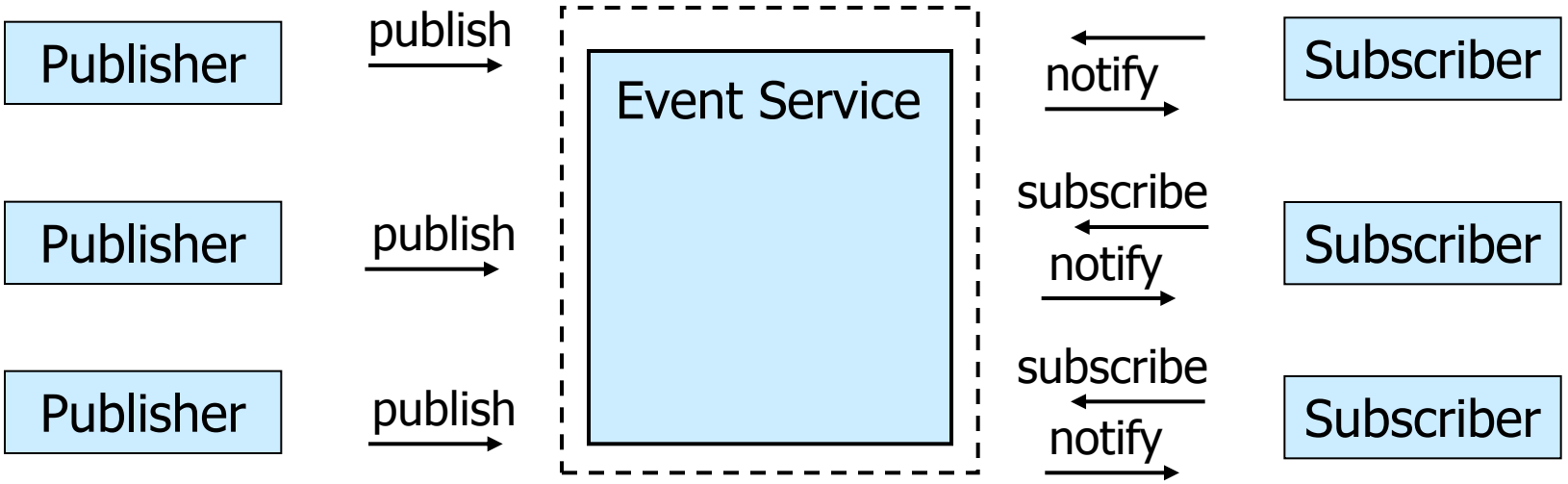
✖ Synchronous request/reply interaction
- Asynchronous Method Invocation (AMI)
- But implementations may not be loosely coupled

✖ 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

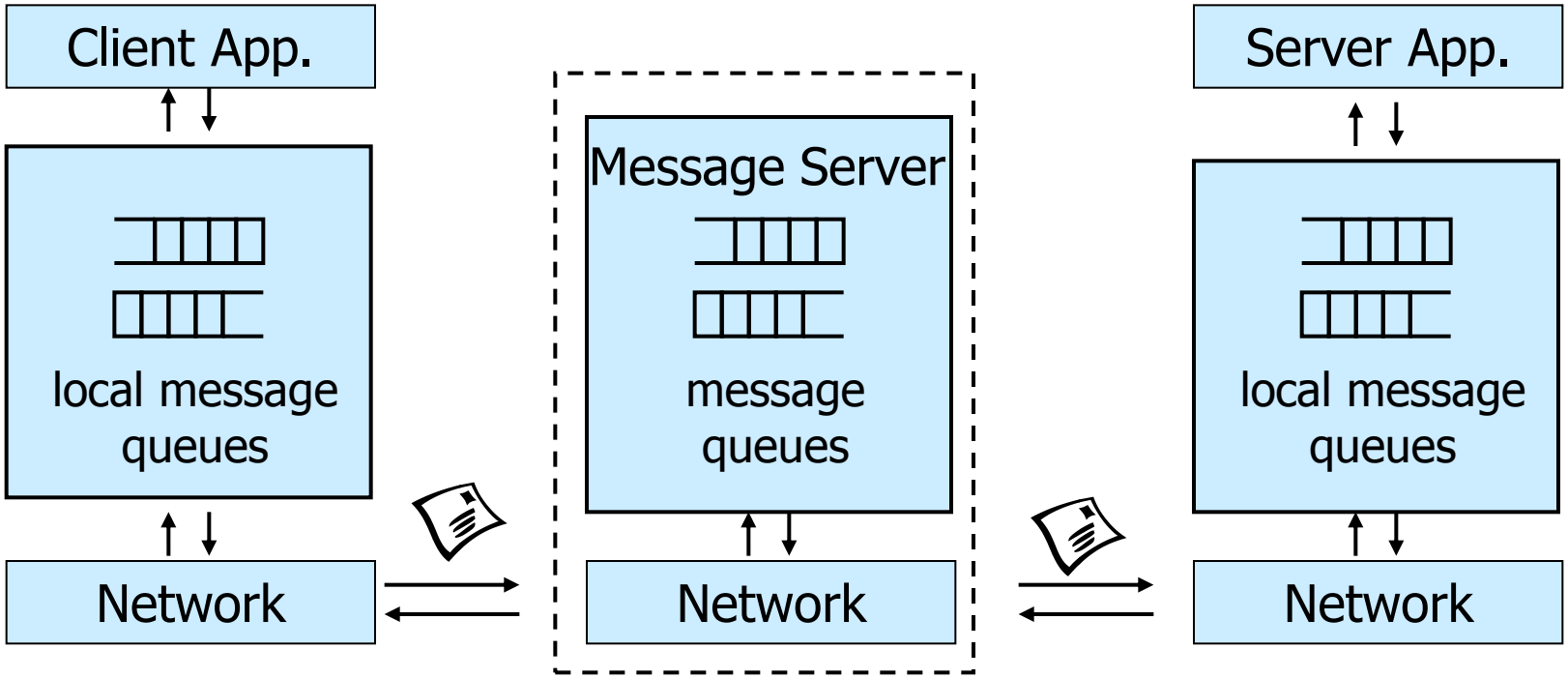


Message Queues aka Message Oriented Middleware (MOM)

- 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

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