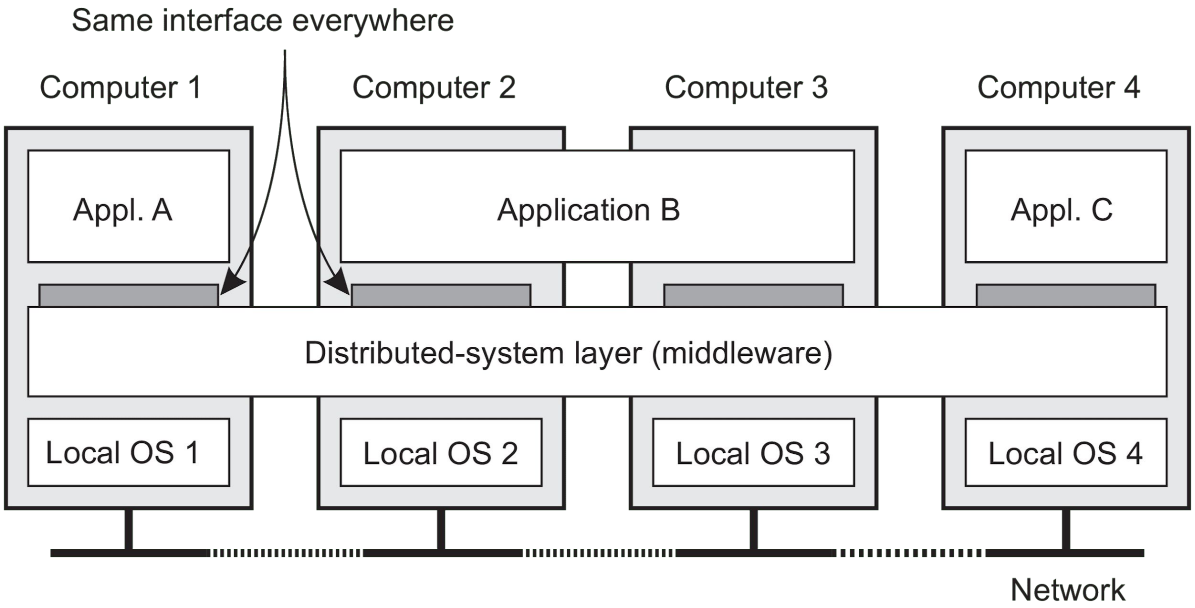
Communication

Agenda

- Remote Procedure Call
- Message-oriented Middleware

References: Chapter 4 of “Distributed Systems by M. van Steen and A. S. Tanenbaum”

Middleware



A software layer that abstracts from heterogeneity of networks, hardware, operating systems and programming languages, and provides a uniform computational and communication model

Services

A middleware offers services to applications, e.g.,

- Communication
- Security mechanism
- Transactions
- Error-recovery
- Management of shared resources

Services are independent of the specific application

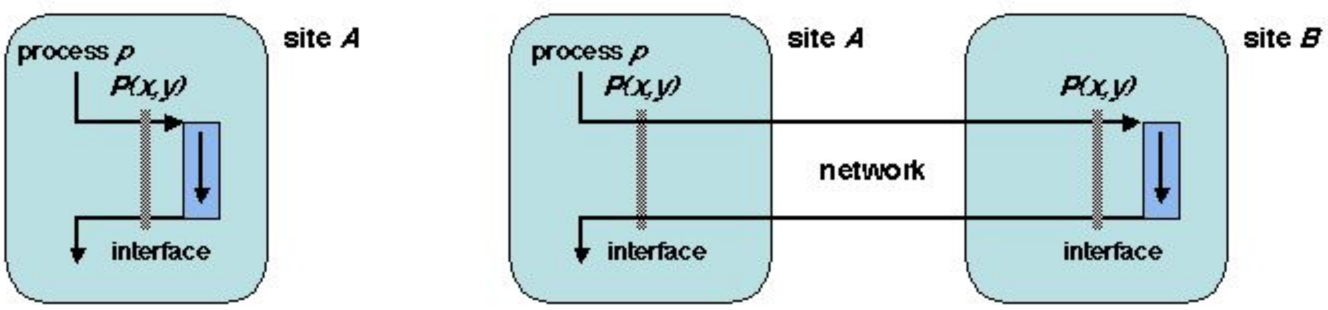
Direct Coordination

A kind of communication where components are

- **Referentially coupled:** during the communication components use an explicit reference to the communication partner (an address, a name)
- **Temporally coupled:** both components must be up and running to communicate

Remote Procedure Call

The idea: Programs can call procedures on other machines

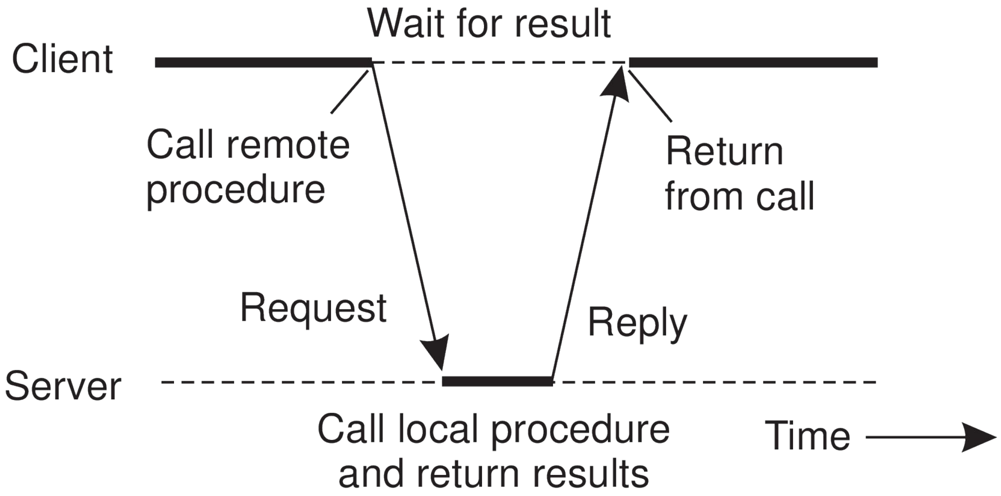


(a) Local call

(b) Remote call

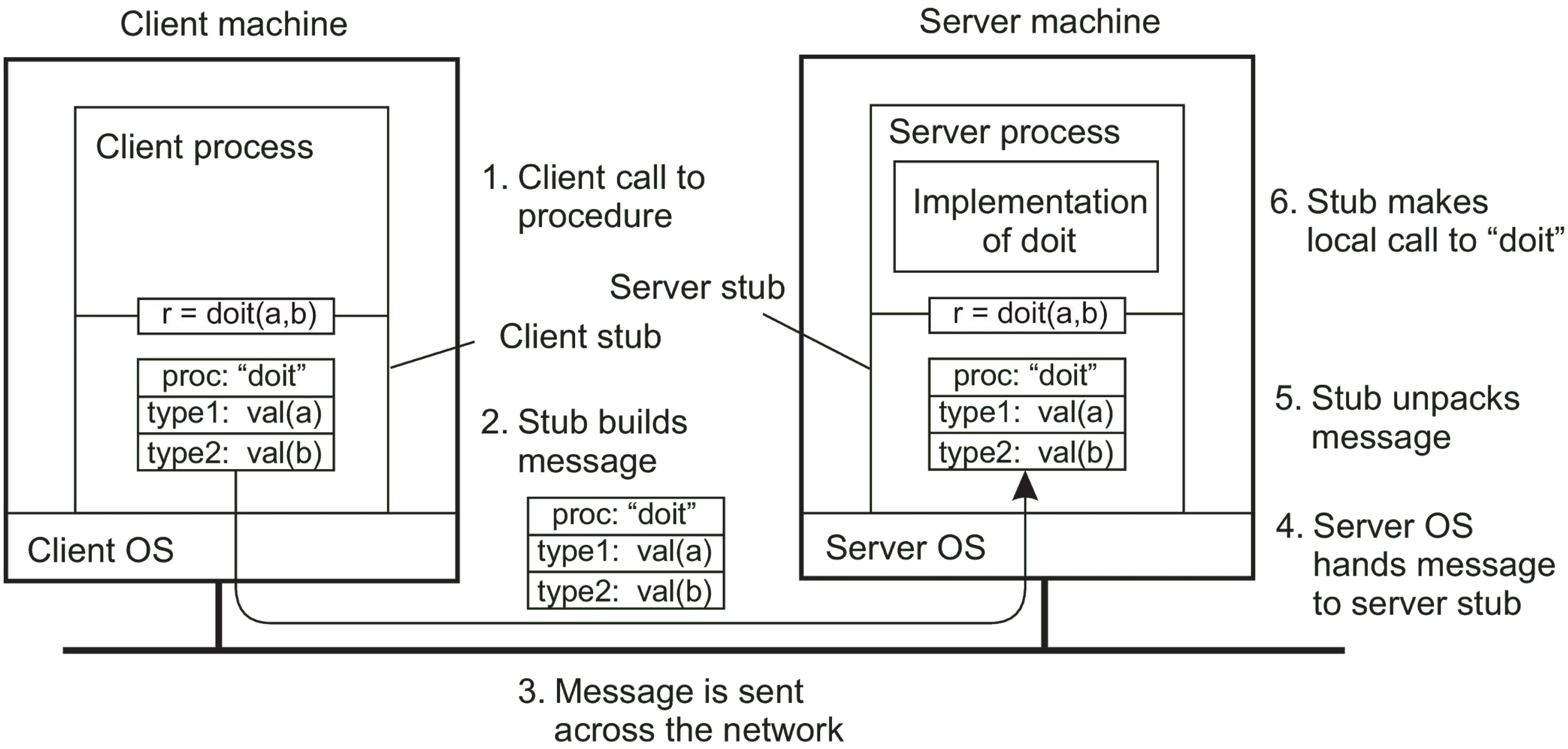
A Glimpse

- When a client calls a remote procedure, its execution is suspended
- A request is send to the server
- The server calls the corresponding local procedure
- The results of the call is returned to the client
- The client resumes the execution



Challenges: two different address spaces, serialization of parameters & return value, management of possible failures

How RPC works



The Steps For Calling a Remote Procedure

1. The client calls the procedure stub
2. The stub builds a message pass it to OS
3. The client OS sends the message to the server
4. The server receives the message and calls the server stub
5. The stub extracts the parameter from the message
6. The stub calls the procedure and gets the result
7. The stub builds a message and pass it to OS
8. The server sends the message to the client
9. The client stub gets the message and unpacks the result
10. The client resumes the execution

A Conceptual Example (Client-side)

```
# Stub on the client
class Client:
  def append(self, data, dbList):
    msglst = (APPEND, data, dbList)
    self.chan.sendTo(self.server, msglst)
    msgrcv =
self.chan.recvFrom(self.server)
    return msgrcv[1]
```

```
# Remote invocation on the client
dbHandle = client.append(newTable,
dbHandle)
```

A Conceptual Example (Server-side)

```
# Actual data structure on the server
class DBList:
    def append(self, data):
        self.value = self.value + [data]
        return self

# Stub on the server
class Server:
    def append(self, data, dbList):
        return dbList.append(data)
```

```
# Main loop of the server
while True:
    msgreq = self.chan.recvFromAny()
    client = msgreq[0]
    msgrpc = msgreq[1]
    if APPEND == msgrpc[0]:
        result = self.append(msgrpc[1],
msgrpc[2])
        self.chan.sendTo(client, result)
```

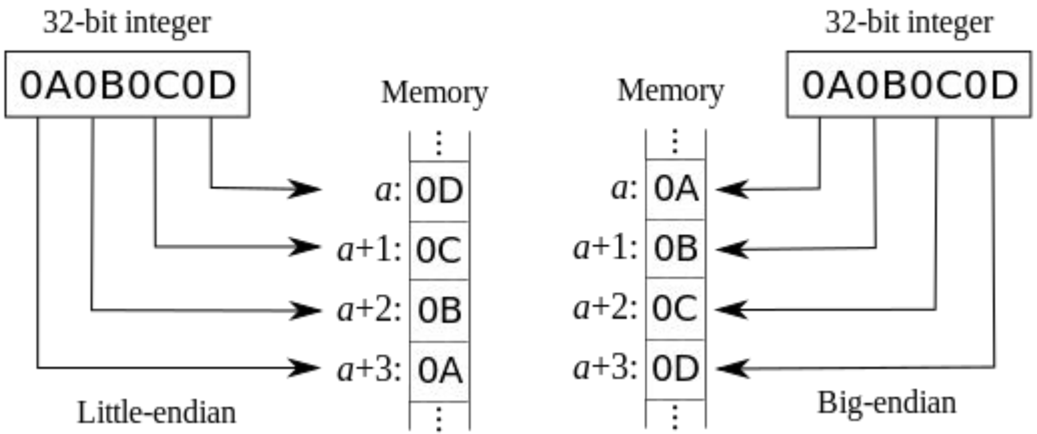
Parameter Passing

- The operation of packing parameters into a message is called **parameter marshaling**
- In the end, a message is a sequence of bytes containing parameters + other information useful for the server
- There are two challenges to solve in parameter passing:
  1. Client and server may have **different data representation**
  2. They have to use the **same encoding**

Data representation

The order of byte in memory may differ between machines

Little-endian vs Big-endian



Data representation

Solution: transform data into a machine-independent format

Network Byte Order

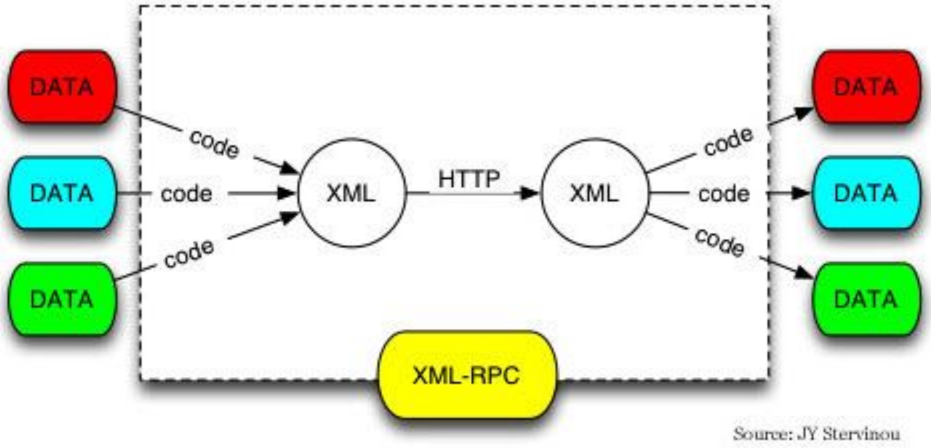
```c
#include <netinet/in.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Data Encoding

Client and server have to agree on how basic/complex data values are represented, e.g.,

- Layout of multi-dimensional arrays: row-major vs column-major
- Layout of records/objects

Example: XML-RPC



XML-RPC uses HTTP as the transport protocol and XML as the encoding format

XML-RPC: Data Encoding

```
<array>
 <data>
  <value><i4>1404</i4></value>
  <value><string>
    Something here
  </string></value>
  <value><i4>1</i4></value>
 </data>
</array>
```

```
<boolean>1</boolean>

<dateTime.iso8601>
  19980717T14:08:55
</dateTime.iso8601>

<double>-12.53</double>

<string>Hello world!</string>
```

XML-RPC: Message Encoding

```xml
<?xml version="1.0"?>
<methodCall>
 <methodName>
  examples.getStateName
 </methodName>
 <params>
  <param>
    <value><i4>40</i4></value>
  </param>
 </params>
</methodCall>
```

```xml
<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
    <value><string>
     South Dakota
    </string></value>
  </param>
 </params>
</methodResponse>
```

Passing References/Pointers

```
# Stub on the client
class Client:
  def append(self, data, dbList):
    msglst = (APPEND, data, dbList)
    self.chan.sendTo(self.server, msglst)
    msgrcv =
self.chan.recvFrom(self.server)
    return msgrcv[1]
```

We cannot simply pass a reference to the server

The Problem

- Client and server have two different address spaces
- References and pointers refer to memory locations that have meaning **locally** only to the calling process
- A reference may refer to something different in the client and in the server

A First Solution

We copy the entire data structure the pointer is referring to

- Quite easy for flat data structure, arrays
- Hard for nested data types, e.g., trees, graphs, etc.
  - Copying large and nested data structure may be inconvenient because of the overhead due to the (un)marshalling and transmission processes
  - Many OO languages provide an automatic support for (un)marshalling objects

Our Conceptual Example Revisited

```
# Stub on the client
class Client:
    def append(self, data, dbList):
        msglst = (APPEND, data, dbList)
        msgsnd = pickle.dumps(msglst)
        self.chan.sendTo(self.server, msglst)
        msgrcv =
self.chan.recvFrom(self.server)
        return msgrcv[1]
```

```
# Main loop of the server
while True:
    msgreq = self.chan.recvFromAny()
    client = msgreq[0]
    msgrpc = pickle.loads(msgreq[1])
    if APPEND == msgrpc[0]:
        result = self.append(msgrpc[1],
msgrpc[2])
        msgres = pickle.dumps(result)
        self.chan.sendTo(client, result)
```

Example: XML-RPC Request

```xml
<?xml version="1.0"?>
<methodCall>
  <methodName> list.multiply </methodName>
  <params> <param>
      <array> <data>
         <value><i4>1404</i4></value>
         <value><i4>10</i4></value>
        <value><i4>1</i4></value>
      </data></array>
   </param></params>
</methodCall>
```

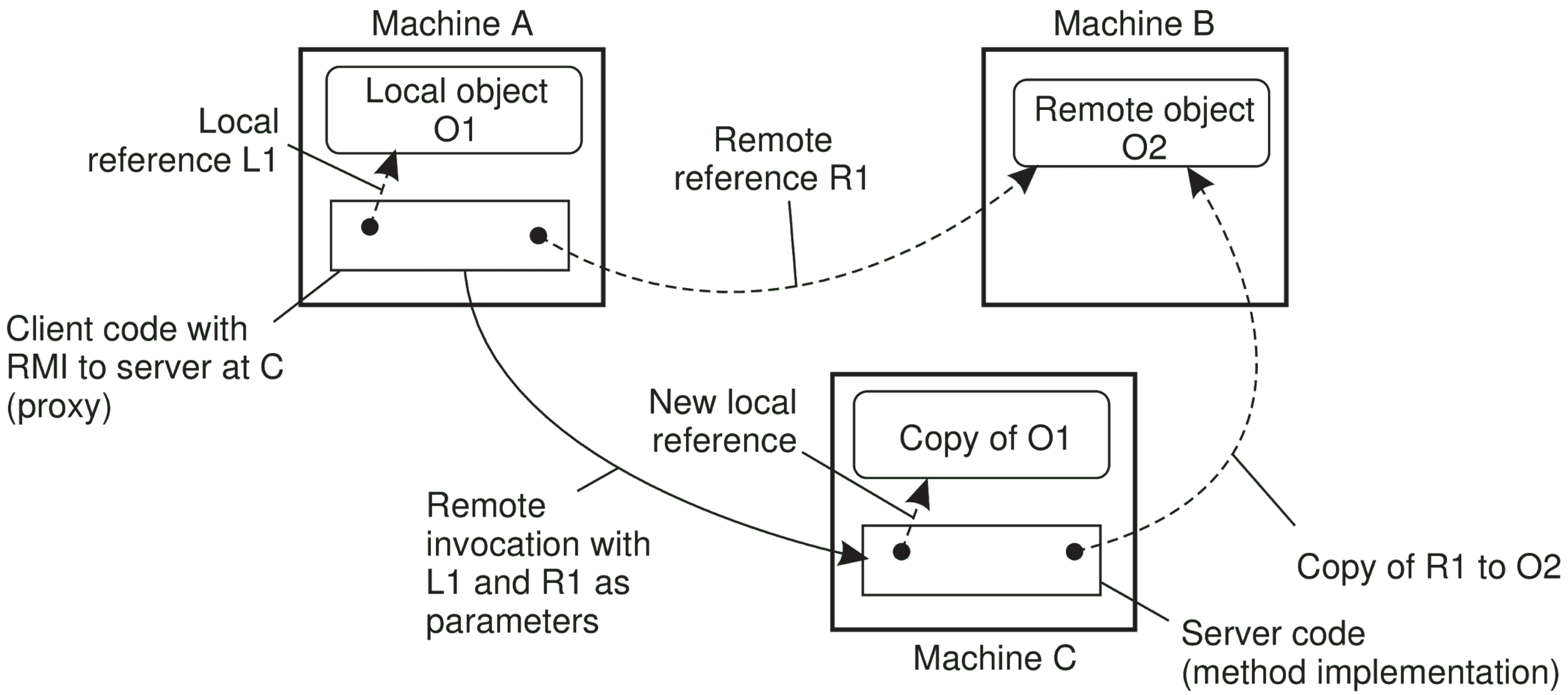
Global References

- References are meaningful to the client and the server
- When they are passed as parameters to a remote procedure, they can be simply copied by one machine to the other
- They are typically used in object-based systems

Parameter Passing in Object-based Systems

- There are two kinds of objects, **local** and **remote**, that are treated differently:
  - Local objects are copied and transmitted entirely
  - For remote objects, only the stub is copied and transmitted
- In Java local and remote objects have different types, i.e., remote objects implements Remote interface

Example



Machine A
Local object
O1
Local
reference L1
Client code with
RMI to server at C
(proxy)
Remote
reference R1
Machine B
Remote object
O2
New local
reference
Copy of O1
Remote
invocation with
L1 and R1 as
parameters
Machine C
Copy of R1 to O2
Server code
(method implementation)

RPC Support

There are two ways in which a RPC mechanism can be provided to developers:

1. As a framework/library together with suitable tools, e.g., a compiler
2. As a programming language construct

RPC Framework

- Programmers need to specify what is remotely exported by providing an **interface** of the service
- Typically, the interface is written in ad hoc language, called **Interface Definition Language** (**IDL**), that needs to be compiled to generate the stubs for the client and the server

Pros: programming language independent

Cons: not fully transparent to programmers

Some Frameworks

Apache Thrift ™

CORBA®

finagle

GRPC

Programming Language-based Support

- The programming language provides constructs or mechanisms to denote a procedure/object as remote
- The compiler of the language takes care of generating the stubs during the compilation

Pros: quite transparent for the programmer

Cons: client & server must be written in the same language

Some Examples

pyro

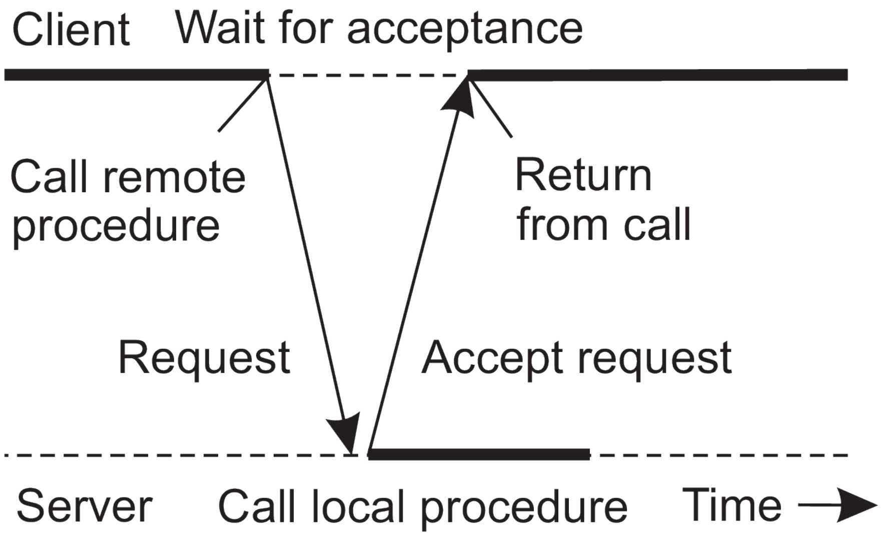
RPyC unbounded computing

Java™

RMI

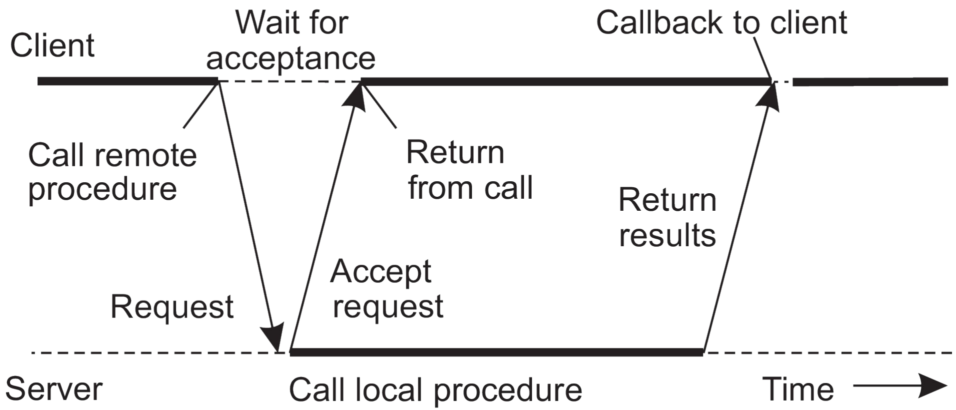
Asynchronous RPC

- After the request the client does not wait for the procedure results but continues its execution and it will wait the response of the server later
- The server immediately sends an acknowledgement as soon as it receives the request from the client



Callbacks

- The client implement a remote procedure to be invoked by the server to return the response
- The client sends the request that include a reference to the procedure, receives the acknowledgement and continues its computation
- The server sends the ack, processes the request and finally invokes the the callback to return the result to the client



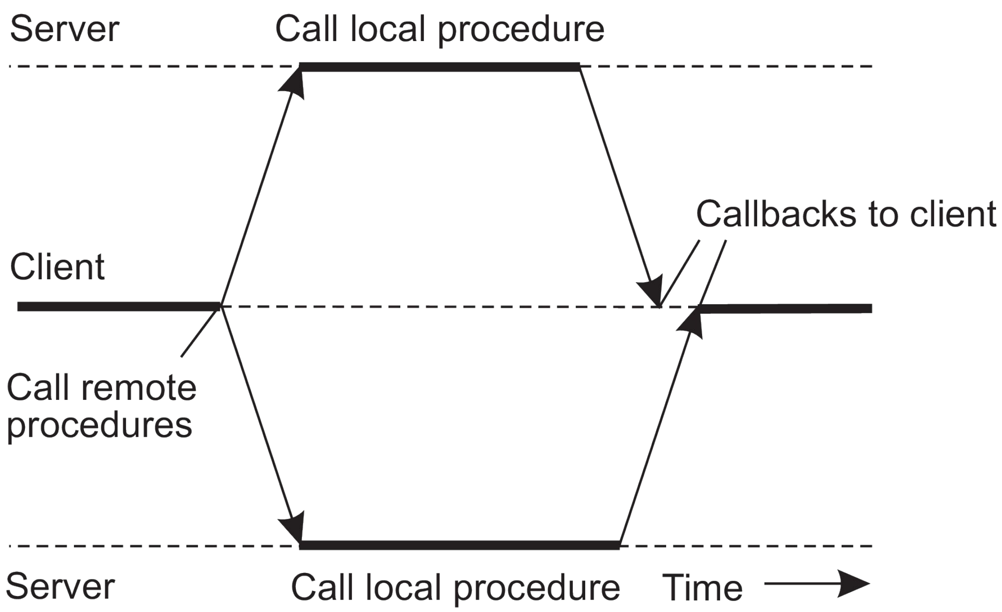
Multicast RPC

One-way RPC: the client does not wait for the acknowledgment of the server

The basic idea: the client sends a request to several servers that process it independently and in parallel

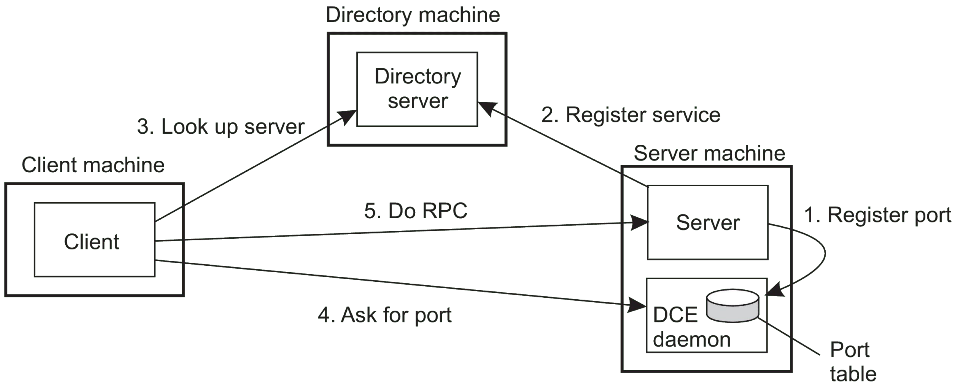
What about the responses?

1. Only the first response that arrives is accepted
2. The responses from all servers are merged together



Binding a Client to a Server

- In real applications there is a preliminary step called "**binding**" that allows a client to take a "reference" to the server
- The **registry** or **directory server** stores information about the servers, usually mapping a symbolic name to the network address of the server
- The server registers to the registry
- The client looks the server up in the registry by using its symbolic name



Example of a List Server: The DB Stub

```
class DBClient:
    def sendrecv(self, message):
        sock = socket()                              # create a socket
        sock.connect((self.host, self.port))         # connect to server
        sock.send(pickle.dumps(message))             # send some data
        result = pickle.loads(sock.recv(1024))       # receive the response
        sock.close()                                 # close the connection
        return result

    ....
```

Example of a List Server: The DB Stub

```
class DBClient:
    def sendrecv(self, message):
    ….
    def create(self):
        self.listID = self.sendrecv([CREATE])
        return self.listID

    def getValue(self):
        return self.sendrecv([GETVALUE, self.listID])

    def appendData(self, data):
        return self.sendrecv([APPEND, data, self.listID])
```

Example of a List Server: The Server

```
class Server:
    def __init__(self, port=PORT):
        # set up the connection with socket, bind, listen
        self.setOfLists = {}        # init: no lists to manage

    def run(self):
        while True:
            (conn, addr) = self.sock.accept()   # accept incoming call
            data = conn.recv(1024)              # fetch data from client
            request = pickle.loads(data)   ….
```

Example of a List Server: The Server

```
class Server:
    def __init__(self, port=PORT):
        ...

    def run(self):
        ...
            if request[0] == CREATE:          # create a list
                listID = len(self.setOfLists) + 1  # allocate listID
                self.setOfLists[listID] = []    # initialize to empty
                conn.send(pickle.dumps(listID))  # return ID
        ....
```

Example of a List Server: The Server

```
class Server:
    def __init__(self, port=PORT):
        ...

    def run(self):
        ...
            elif request[0] == APPEND:          # append request
                listID = request[2]             # fetch listID
                data = request[1]               # fetch data to append
                self.setOfLists[listID].append(data) # append it to the list
                conn.send(pickle.dumps(OK))     # return an OK
            ....
```

Example of a List Server: The Server

```
class Server:
    def __init__(self, port=PORT):
        ...

    def run(self):
        ...
            elif request[0] == GETVALUE:        # read request
                listID = request[1]             # fetch listID
                result = self.setOfLists[listID]  # get the elements
                conn.send(pickle.dumps(result))   # return the list
                conn.close()                    # close the connection
```

Example of a List Server: 2 Clients

```
client1 = Client(CLIENT1)                            # create client
dbClient1 = DBClient(HOST,PORT)                      # create reference
dbClient1.create()                                   # create new list
dbClient1.appendData('Client 1')                     # append some data
client1.sendTo(HOSTCL2,CLIENT2,dbClient1)            # send to other client
```

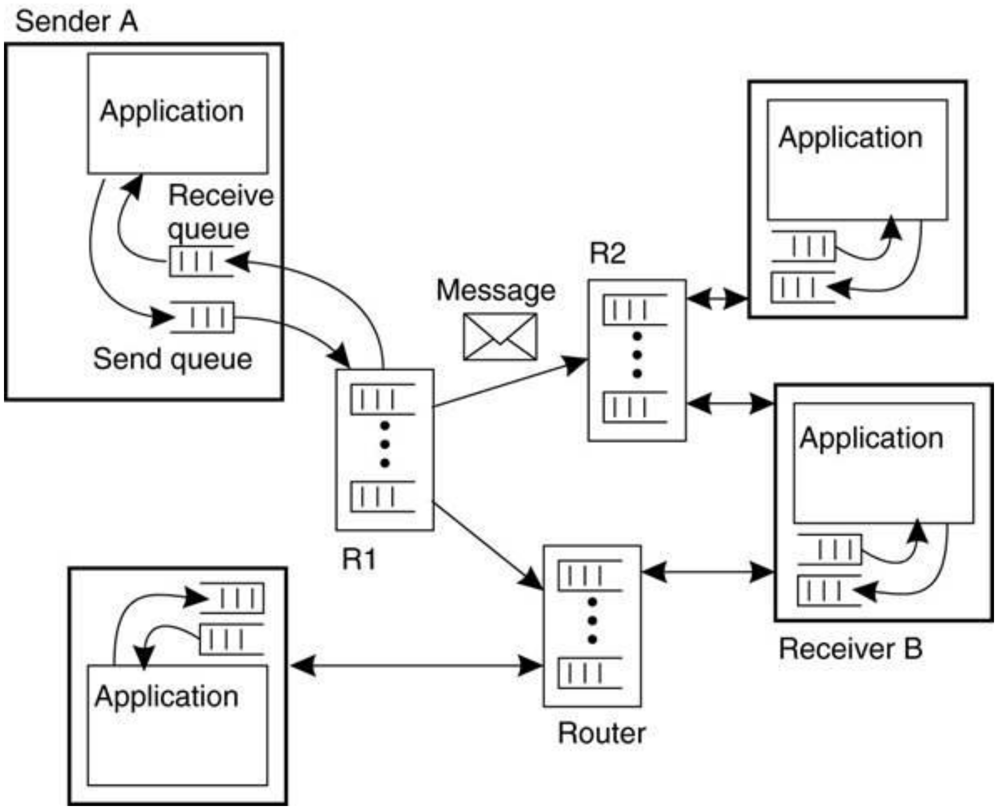
```
client2 = Client(CLIENT2)           # create a new client
data = client2.recvAny()            # block until data is sent
dbClient2 = pickle.loads(data)      # receive reference
dbClient2.appendData('Client 2')    # append data to same list
```

Message-oriented Middleware

The idea: components communicates by exchanging messages with each other

Persistent asynchronous communication: sender & receiver do not need to be active during the transmission of messages, the middleware provide storage services

Message-queueing Model



Message-queueing Model

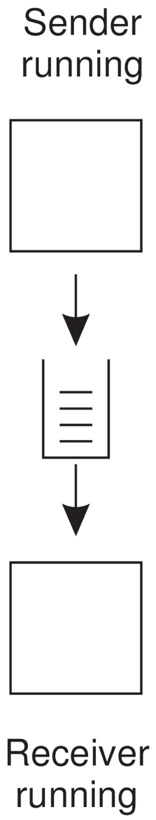
- Each application is equipped with a **local** queue that contains the sent and received messages
- A queue may be shared by multiple applications
- Messages from the sender queue are forwarded over a series of communication servers until the destination queue is reached

Communication Properties

- Communication happens only by inserting/removing messages from queues
- Communication is loosely coupled in time
  - There is no need for the receiver to be running when a message is sent
  - A message remains in the queue until it is explicitly removed

Example

- The sender & the receiver are running at the same time
- The receiver can start receiving the message while the sender is sending it

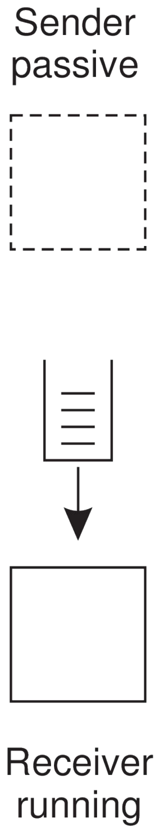


Example

- The receiver is not ready for receiving messages
- The sender can send messages that will be stored in the queue

Example

- The sender is not sending, but there are messages in the queue
- The receiver get the messages from the queue



Example

- The sender & receiver are not involved in communication
- The queue is “running” and stores messages sent until now

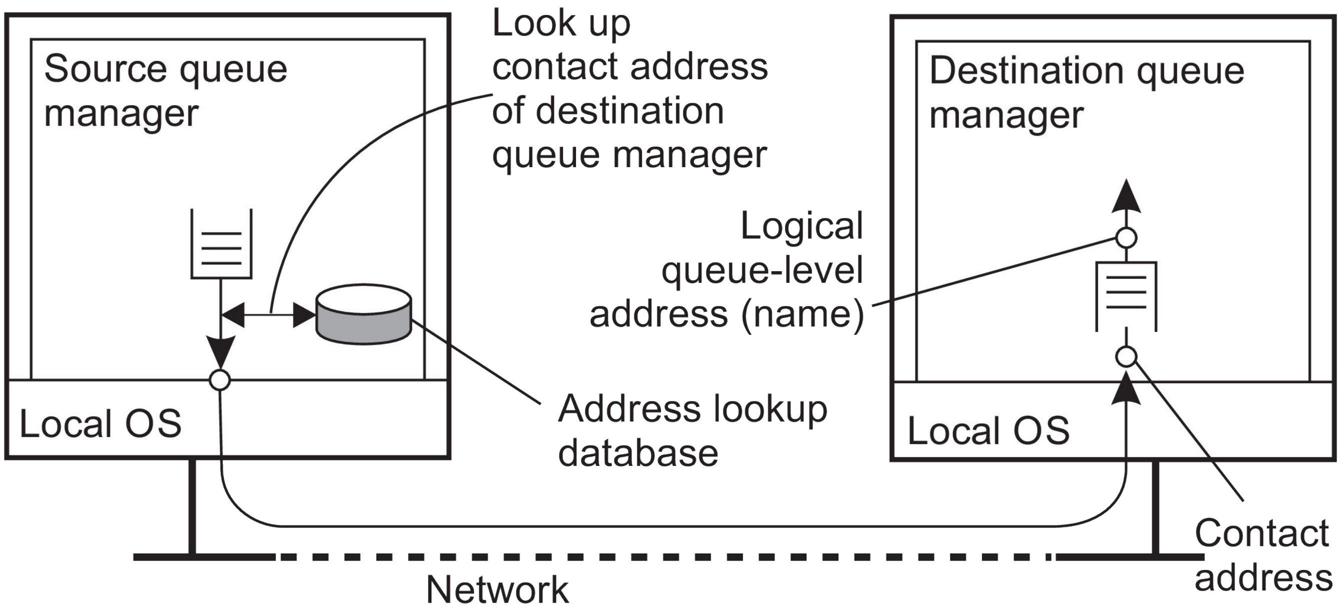
Messages

- Messages can contain any kinds of data and may be of any size within a given limit
- The middleware can support automatically fragmentation of messages
- A destination is identified together with its queue by a systemwide name
- Each message contains the address/name of the destination

Conceptual MoM API

put	Append a msg to a queue
get	Remove the first msg from the queue (blocking)
poll	Remove the first msg from the queue (non-blocking)
notify	Install an handler to be called when a msg arrives

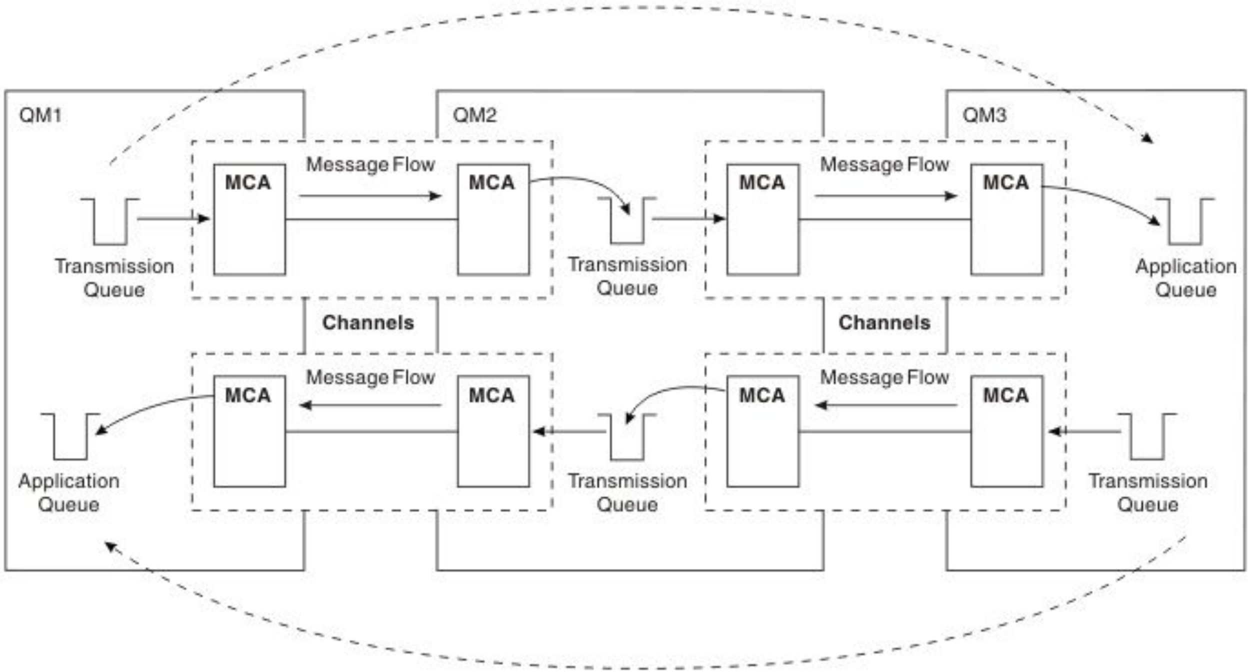
Queue Managers



Queue Managers

- It can be implemented either as a library linked with the application or as a separate process
- Each application is associated with a local queue, hence, it is paired with a queue manager
- An application can send/receive messages to/from its local queue
- **Note:** to support persistent asynchronous communication queue managers should be implemented as separate processes

Overlay Network



Queue managers form an overlay network used to forward messages in the system

Addresses

- **Recall:** each queue has a logical name/address that identifies the queue inside the system
- Logical names must be associated to a network address (host, port)
- Each queue manager maintains a translation table mapping names to network addresses
- The table may contain further information

Managing a Messaging Queue System

- An important task in setting up a MQ system is connecting together the queue managers into a overlay network
  - Setting up communication channels
  - Filling and handling the routing and translation tables
- A further effort consists in maintaining the overlay network when the system has to change, e.g., new queue managers

Heterogeneity

- Usually, MoM does not specify the format of messages
- Two different applications can communicate if they use the same protocol (as in network)
- When an application **A** joins the system, we need to ensure that any other application **B** interacting with **A** uses the same protocol
- **Idea:** provide a protocol for each **B** (no, it does not scale)

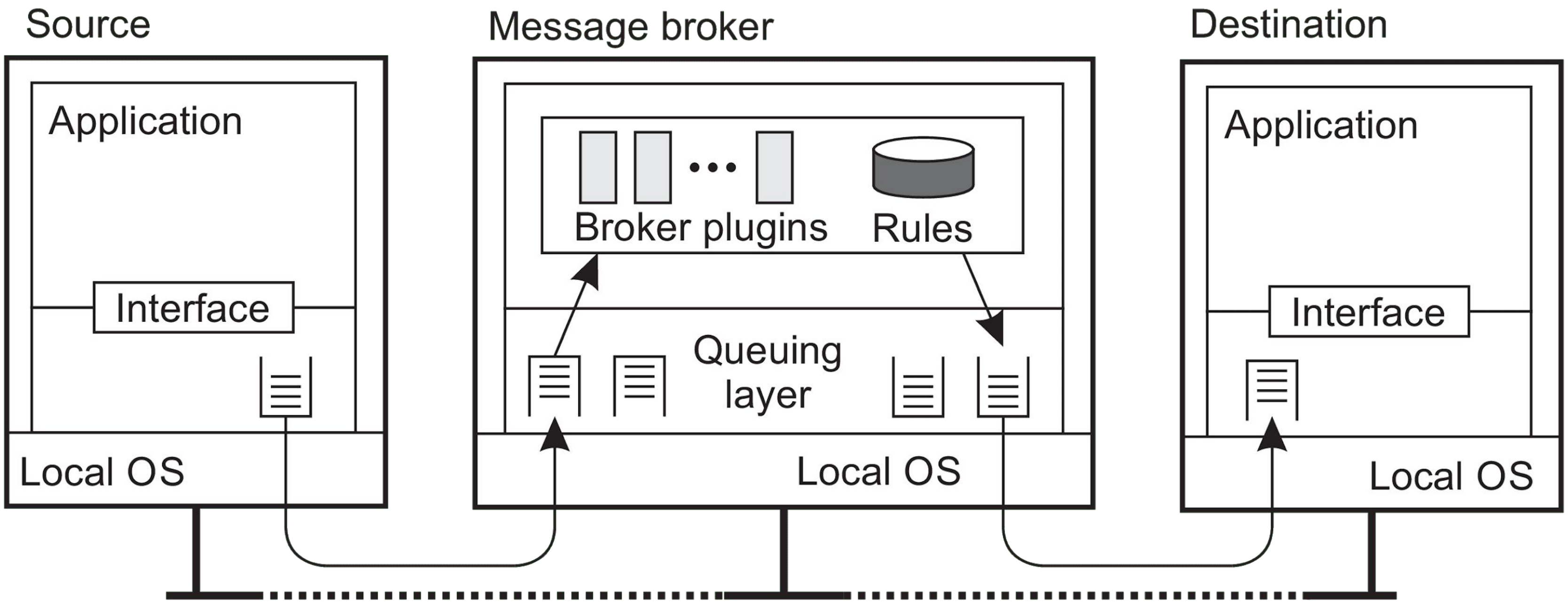
Message Brokers

A component in the MoM system that works as a gateway

1. It converts messages so they can be understood by different applications
2. It matches receivers in a publish-subscribe kind of interaction

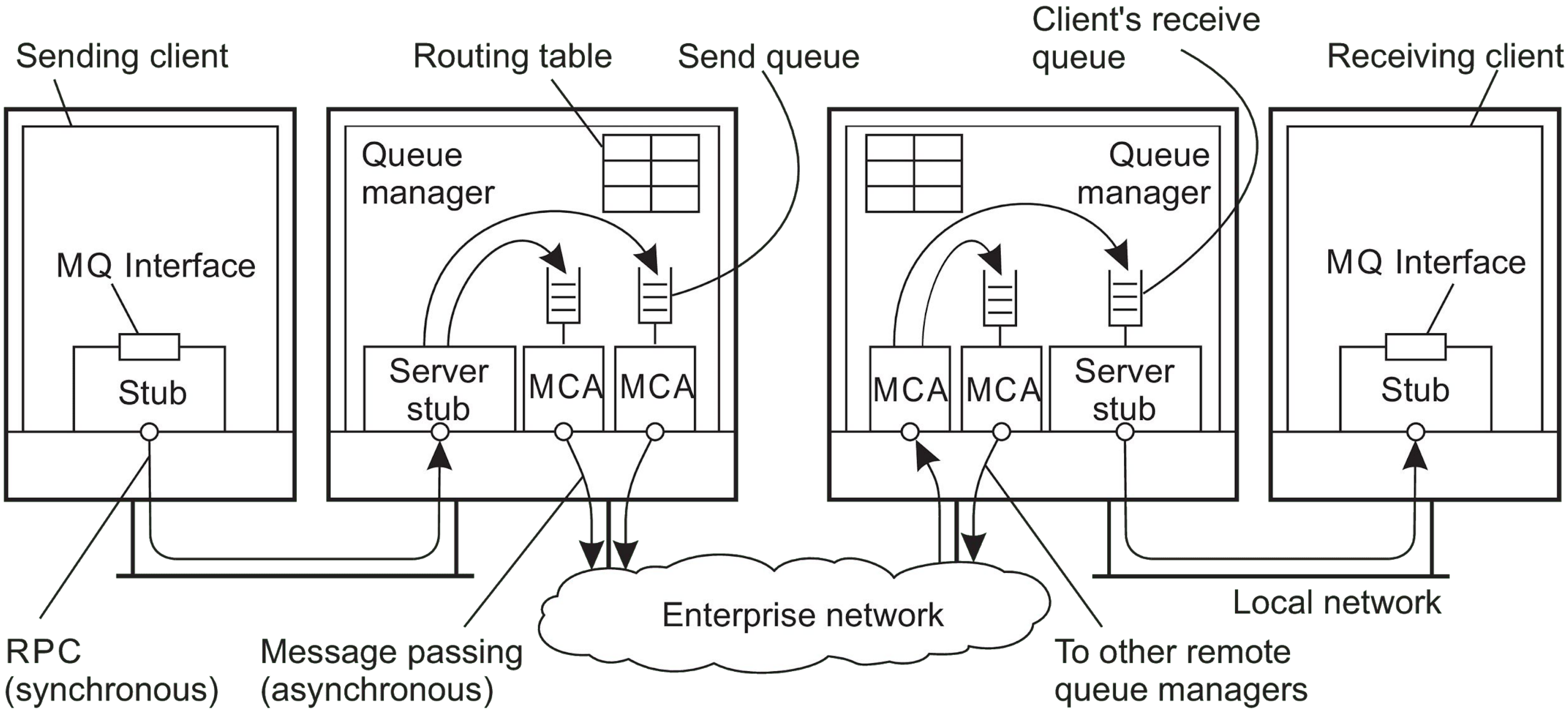
Some Brokers

Message Brokers



There are rules that drive the broker to handle messages

Example: IBM WebSphere



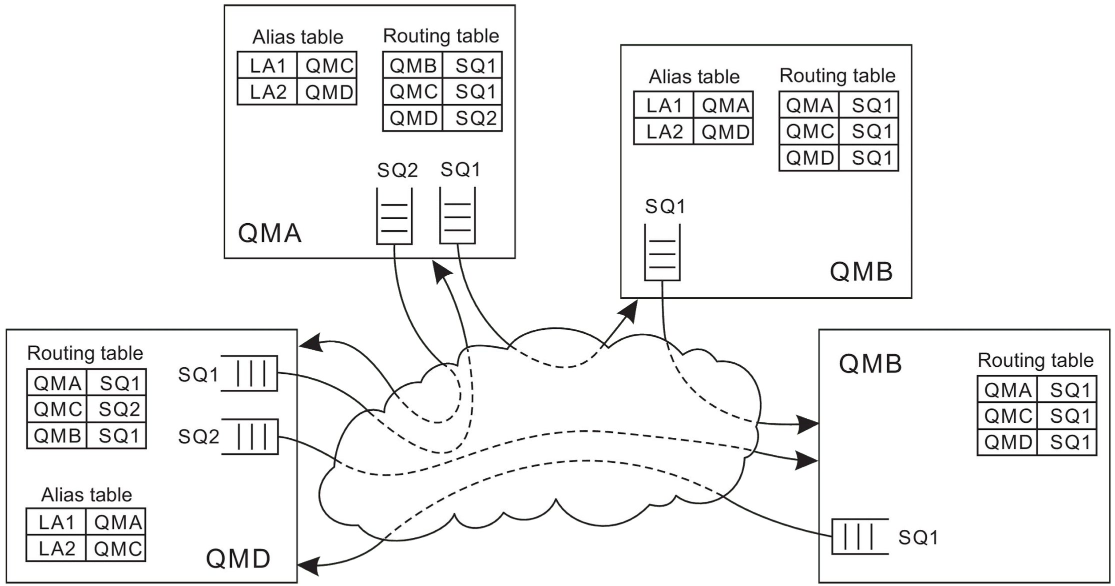
WebSphere Features

- Queue Managers are pairwise connected
- Message channels are unidirectional
- Queue Managers can be:
  - Linked to the application
  - External processes
- The application interacts with its manager through an API that is the same independently of the kind of its Queue Managers

Addressing in WebSphere

- Each message has a header including the name of the destination and the name of the local queue (sender)
- An address is a pair (queue manager, destination)
- Each Queue Manager has a unique name as well as each queue does
- There can be local aliases for denoting Queue Managers and queues

Routing in WebSphere

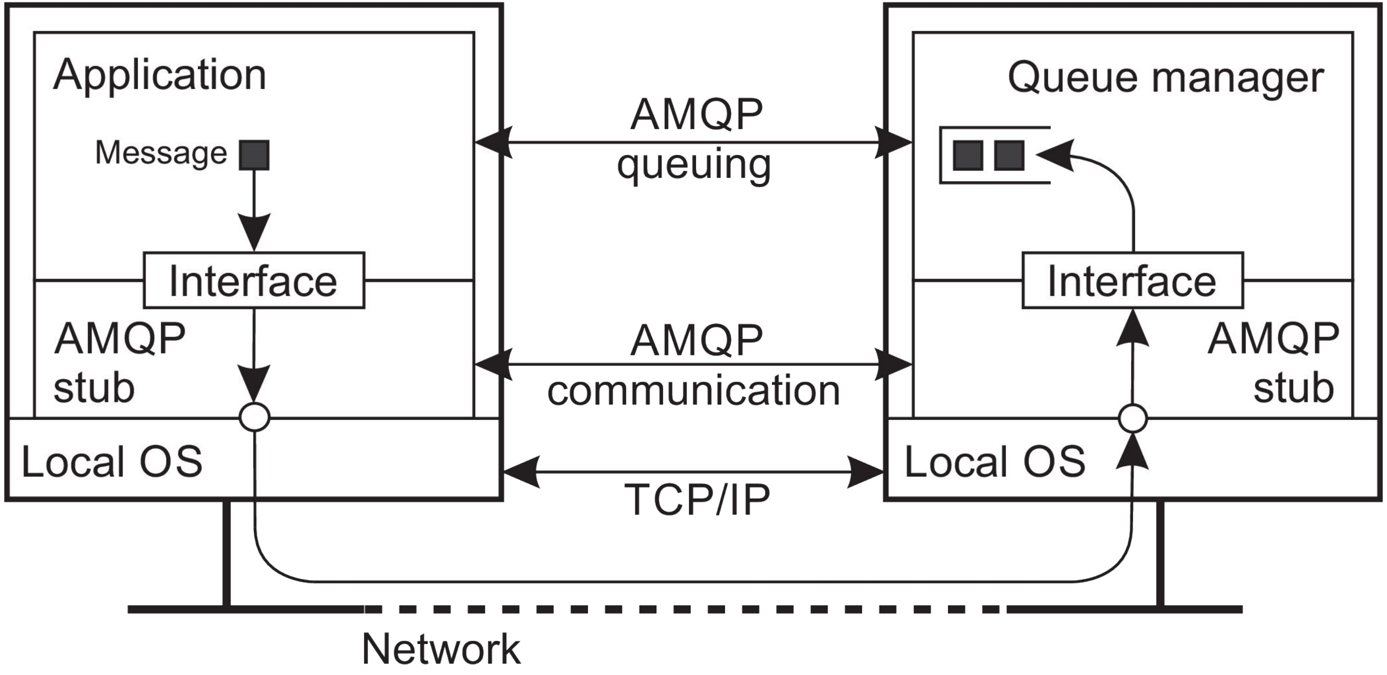


Each Queue Manager has a routing table

Application Interface

MQOPEN	Open a (possible remote) queue
MQCLOSE	Close a queue
MQPUT	Put a message into a opened queue
MQGET	Get a message from a queue

Example: AMQP



An OASIS Standard that allows MoM of different vendors to cooperate

Conclusion

- Remote Procedure Call
- Message-oriented Middleware

References: Chapter 4 of “Distributed Systems by M. van Steen and A. S. Tanenbaum”