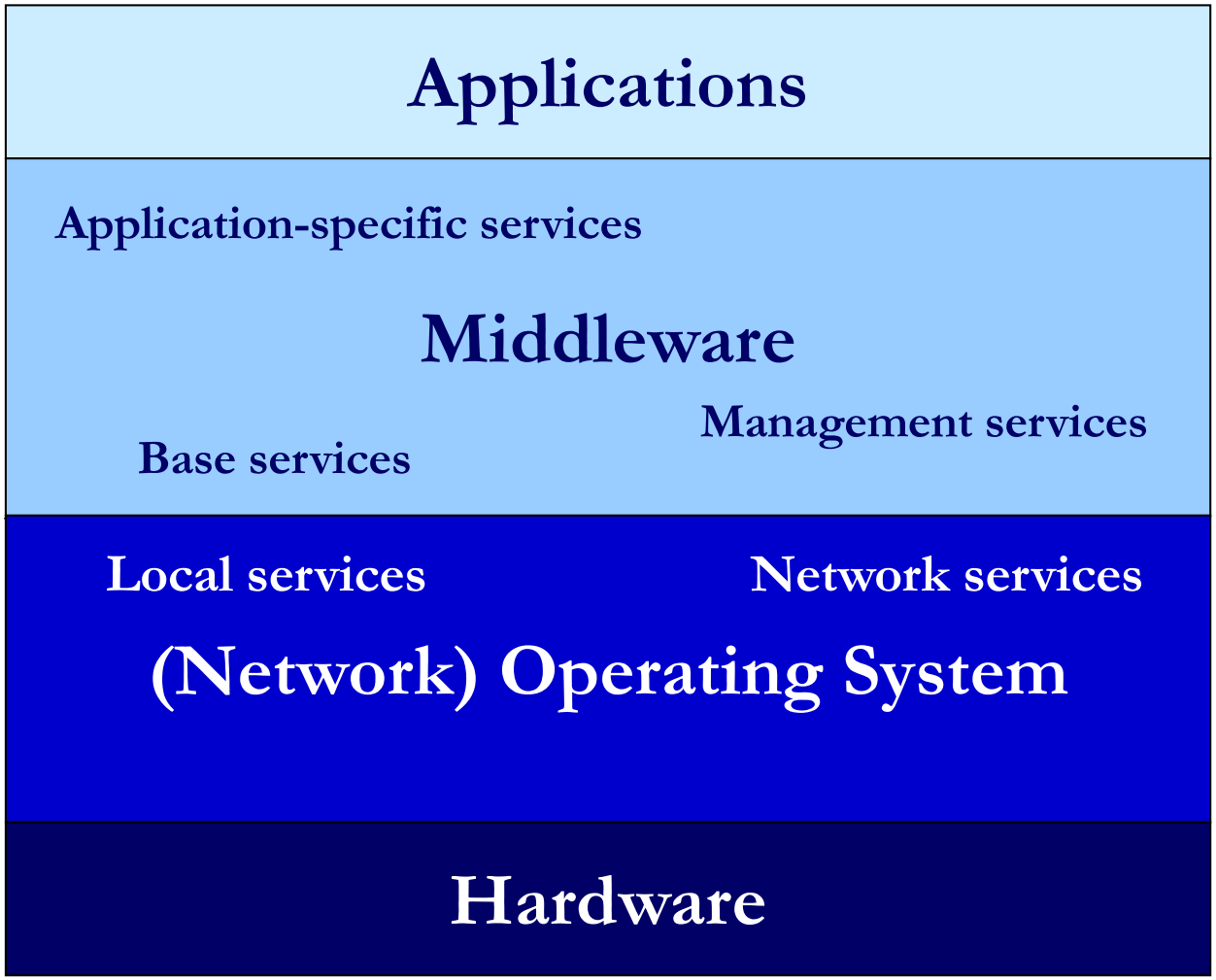
Distributed Systems: Middleware

Gian Pietro Picco
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento, Italy
gianpietro.picco@unitn.it
http://disi.unitn.it/~picco

The Role of Middleware

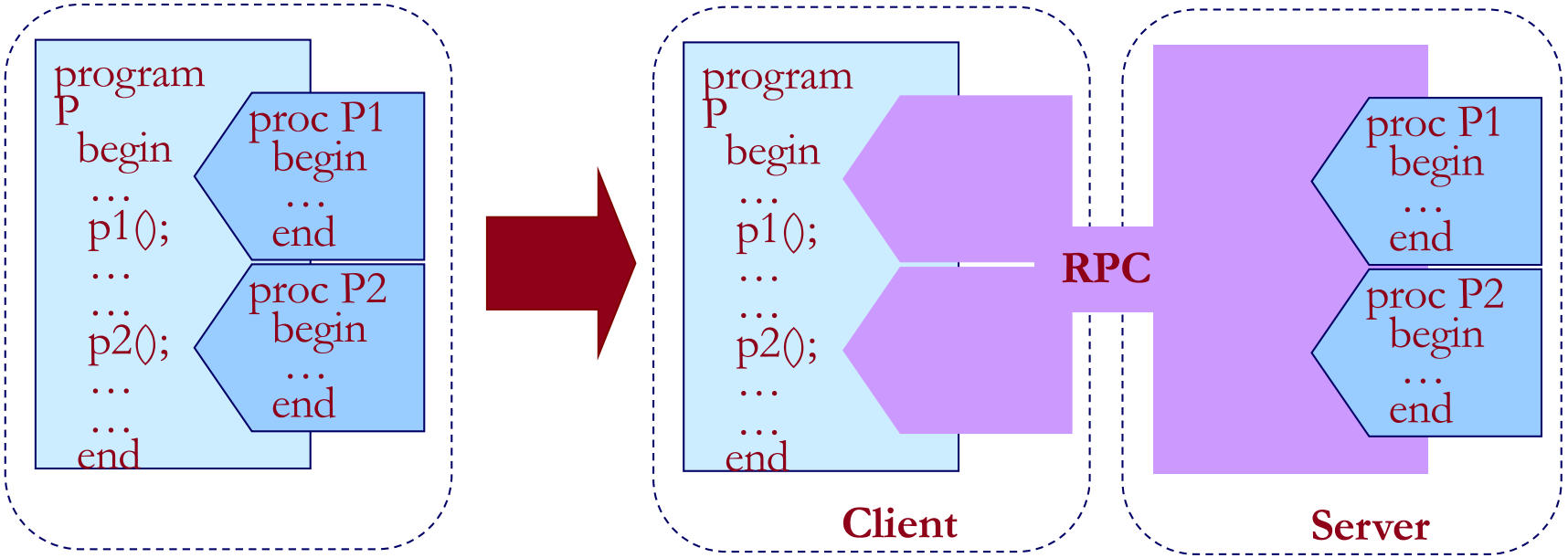


Some Definitions of Middleware

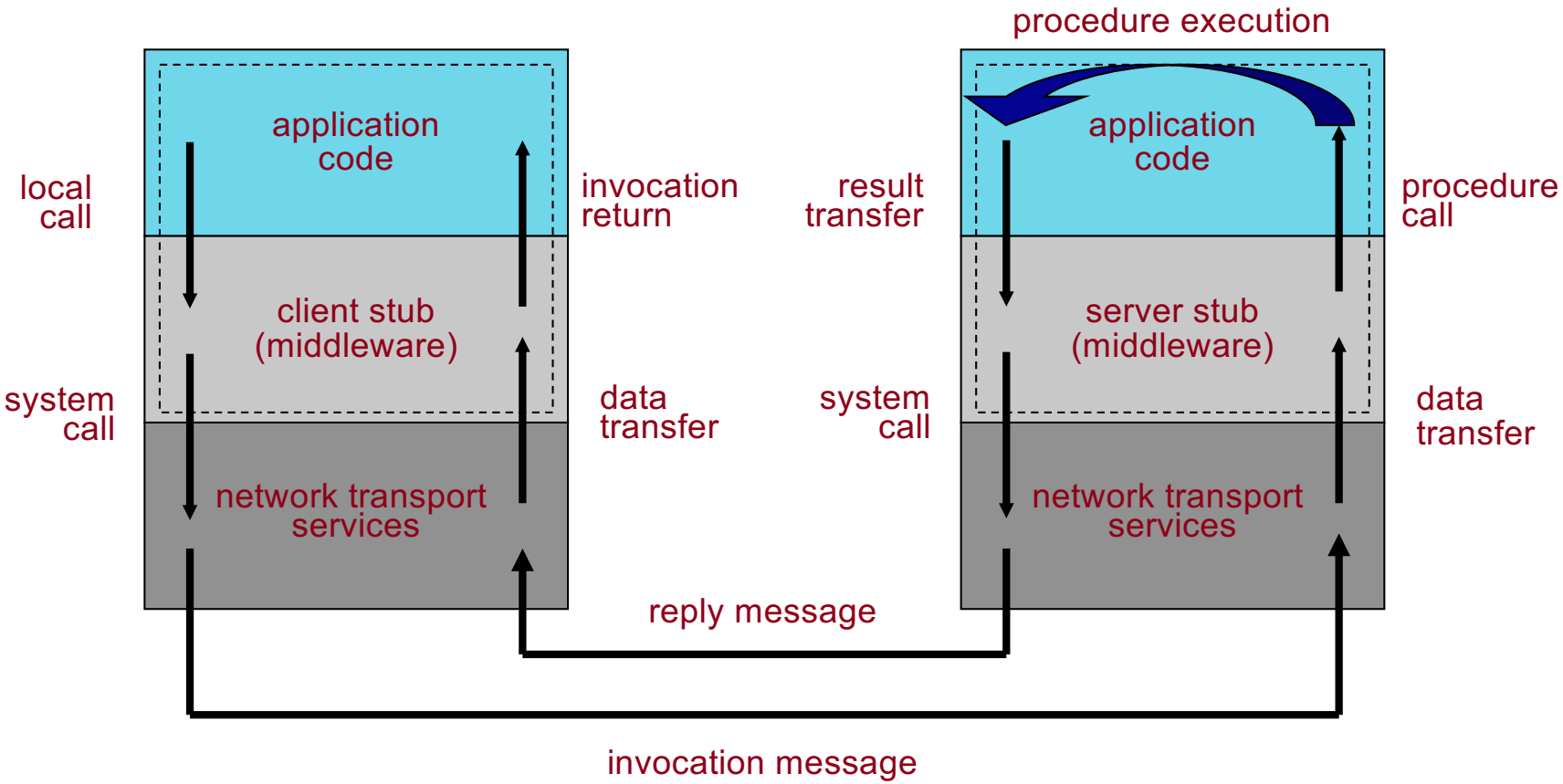
- “Middleware is the slash in the term client/server”
- “Middleware is software that allows elements of applications to interoperate across network links, despite differences in underlying communication protocols, system architectures, OSes, databases, and other application services”
- “Middleware succeeds by providing five main services: hardware independence, interchangeability of key software components (e.g., DBMSs), network independence, operational savings, and administrative savings”
- “Middleware is software that no one wants to pay for”

Remote Procedure Call

- Invented at Sun Microsystems in the early 80s
- ***Problem:*** client-server interaction is handled through the OS primitives for I/O ➔ difficult to develop applications
- ***Idea:*** enable remote access through the well-known procedure call programming



How Does It Work



Parameter Passing—1: Marshalling and Serialization

- Passing a parameter ***value*** poses two problems:
  - Structured data (e.g., structs/records, objects) must be ultimately flattened in a byte stream
    - Called ***serialization*** (or *pickling*, in the context of OODBMSs)
  - Hosts may use different data representations (e.g., little endian vs. big endian, EBCDIC vs. ASCII) and proper conversions are needed
    - Called ***marshalling***
- Middleware provides automated support:
  - The marshalling and serialization code is automatically generated from and becomes part of the stubs
  - Enabled by:
    - A language/platform independent representation of the procedure's signature, written using an ***Interface Definition Language*** (IDL)
    - A data representation format to be used during communication

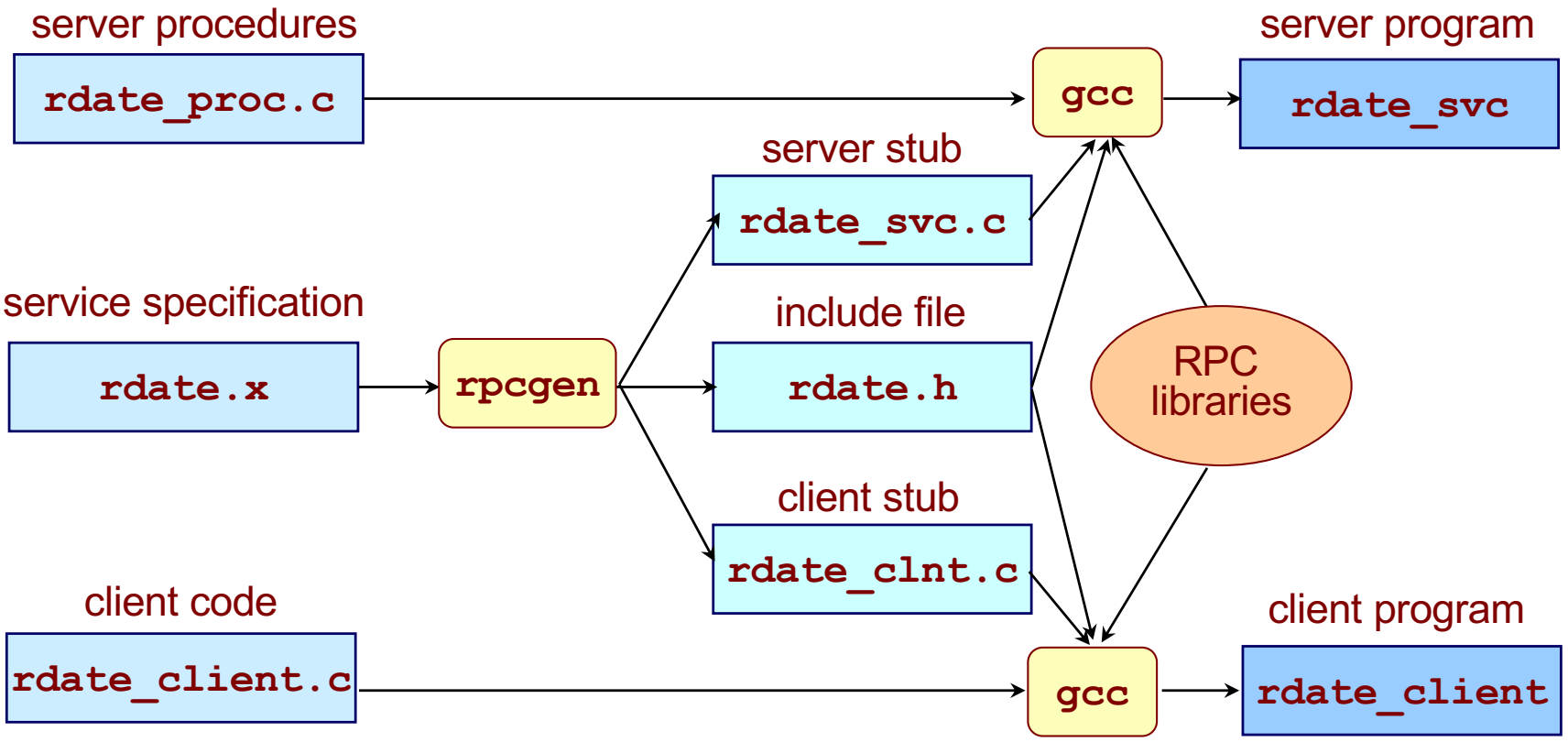
The Role of IDL

- The ***Interface Definition Language (IDL)*** raises the level of abstraction of the service definition
  - It separates the service ***interface*** from its ***implementation***
  - The language comes with “mappings” onto target languages (e.g., C, Pascal, Python…)
- Advantages:
  - Enables the definition of services in a language-independent fashion
  - Being defined formally, an IDL description can be used to ***automatically*** generate the service interface code in the target language

rdate.x

```
program RDATE_PROG {
  version RDATE_VERS {
    long   BIN_DATE(void) = 1; /* proc id */
    string STR_DATE(long) = 2; /* proc id */
  } = 1;                       /* version id */
} = 0x20000001;                /* program id */
```

Sun RPC: Development Cycle



rdate_proc.c

```
#include <time.h>
#include "rdate.h"

long * bin_date_1_svc(void *argp, struct svc_req *reqp) {
  static long  result;
  result = (long) time(NULL);
  return &result;
}

char ** str_date_1_svc(long *argp, struct svc_req *reqp) {
  static char * result;
  result = ctime((time_t *) argp);
  return &result;
}
```

rdate_client.c

```
#include <stdio.h>
#include "rdate.h"
int main (int argc, char *argv[]) {
  CLIENT *clnt; long  *t; char **str;
  if (argc < 2)
    { printf ("usage: %s server_host\n", argv[0]); exit (1); }
  clnt = clnt_create(argv[1], RDATE_PROG, RDATE_VERS, "udp");
  if (clnt == NULL)
    { clnt_pcreateerror(argv[1]); exit(1); }
  t = bin_date_1((void*)NULL, clnt);
  if (t == (long *) NULL)
    { clnt_perror(clnt, "call failed"); }
  str = str_date_1(t, clnt);
  if (str == (char **) NULL)
    { clnt_perror (clnt, "call failed"); }
  printf("Date at host %s is %s\n", argv[1], *str);
  clnt_destroy (clnt);
  exit (0);
}
```

Parameter Passing—2

- How to pass a parameter by reference?
  - Many languages do not provide a notion of reference, but only of pointer
  - A pointer is meaningful only within the address space of a process…
- Often, this feature is simply not supported
- Otherwise, a possibility is to use call by value/result instead of call by reference
  - Semantics is different!
  - Works with arrays but not with arbitrary data structures
  - Optimizations are possible if input- or output-only

Sun RPC (ONC RPC)

- Sun Microsystems' RPC (also called Open Network Computing RPC, ONC RPC) is the *de facto* standard over the Internet
  - At the core of NFS, and many other services
  - Found in modern Unix systems (e.g., Linux)
- Data format specified by XDR (eXternal Data Representation)
  - Initially only for data representation, then extended in a proper IDL
- Transport can use either TCP or UDP
- Parameter passing:
  - only pass by copy is allowed (no pointers)
  - only one input and one output parameter
- Provision for DES security

DCE RPC

- The Distributed Computing Environment (DCE) is a set of specifications and a reference implementation
  - From Open Software Foundation (OSF), later renamed as Open Group, no-profit standardization organization
- Several invocation semantics are offered
  - At most once, idempotent, broadcast
- Several services are provided on top of RPC:
  - Directory service
  - Distributed time service
  - Distributed file service
- Security is provided through Kerberos
- Microsoft's DCOM is based on DCE
- Recently extended towards distributed objects

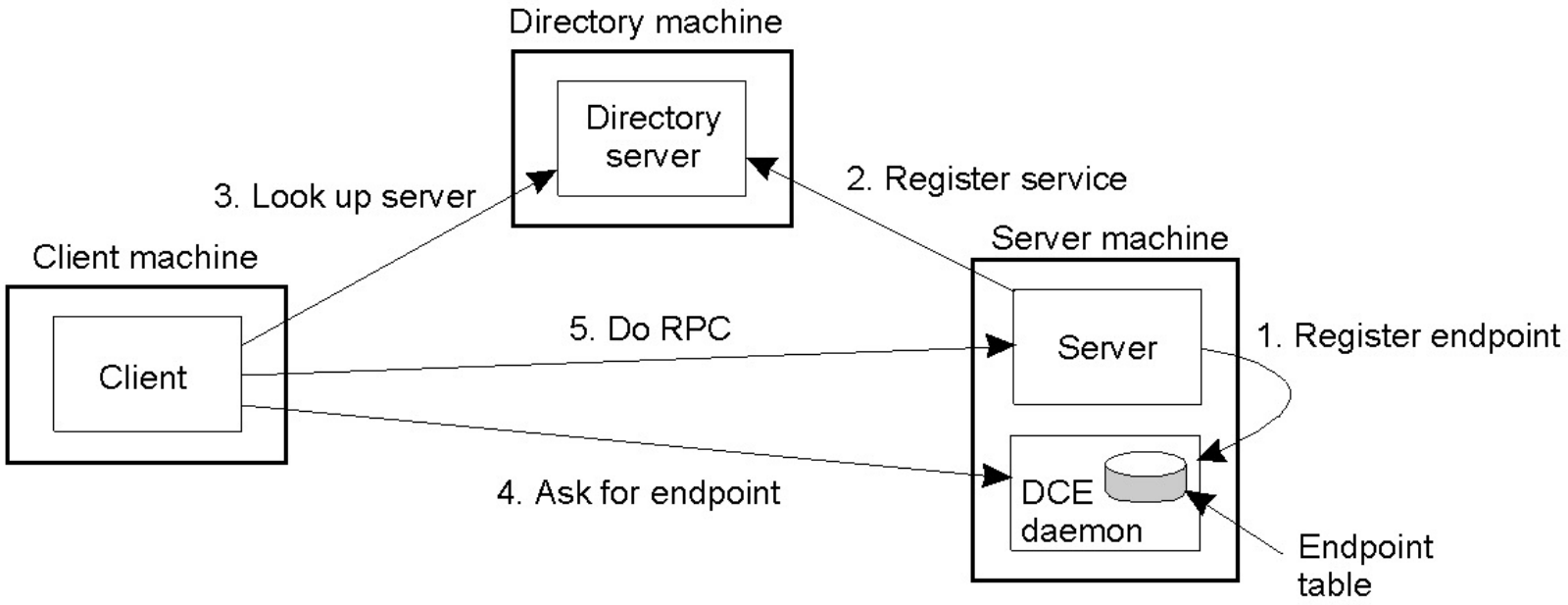
Dynamic Binding

- ***Problem:*** find out which server (process) provides a given service
  - Hard-wiring this information in the client code is highly undesirable
  - Two distinct problems:
    1. Find out ***where*** the server process is
    2. Find out how to establish communication with it

Sun's Solution

- Introduce a daemon process (**`portmap`**) that binds calls and server/ports:
  - The server picks an available port and tells it to **`portmap`**, along with the service identifier
  - Clients contact a given **`portmap`** and request the port necessary to establish communication
- **`portmap`** provides its services only to local clients, i.e., it solves only the second problem
  - The client must know in advance ***where*** the service resides
  - However:
    - a client can multicast a query to multiple daemons
    - more sophisticated mechanisms can be built or integrated
      - e.g., directory services

DCE's Solution



- The DCE daemon works like **`portmap`**
- The directory server (aka ***binder*** daemon) enables location transparency:
  - Client need not know in advance where the service is: they only need to know where the directory service is
  - In DCE, the directory service can actually be distributed
    - To improve scalability over many servers
  - Step 3 is needed only once per session

Dynamic Activation

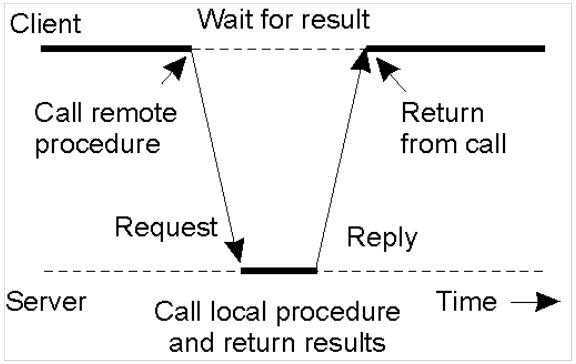
- ***Problem:*** server processes may remain active even in absence of requests, wasting resources
- ***Solution:*** introduce another (local) server daemon that:
  - Forks the process to serve the request
  - Redirects the request if the process is already active
  - Clearly, the first request is served less efficiently
- In Sun RPC:
  - **`inetd`** daemon
  - the mapping between requested service and server process is stored in a configuration file (**`/etc/services`**)

Lightweight RPC

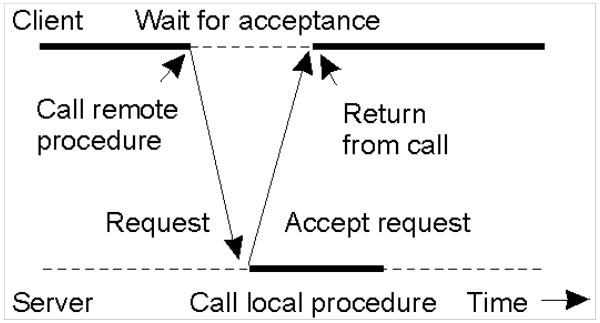
- It is natural to use the same primitives for inter-process communication, regardless of distribution
  - But using conventional RPC would lead to wasted resources: no need for TCP/UDP on a single machine!
- Lightweight RPC: message passing using local facilities
  - Communication exploits a private shared memory region
  - Lightweight RPC invocation:
    - Client stub copies the parameters on the shared stack and then performs a system call
    - Kernel does a context switch, to execute the procedure in the server
    - Results are copied on the stack and another system call + context switch brings execution back to the client
  - Advantages:
    - uses less threads/processes (no need to listen on a channel)
    - 1 parameter copy instead of 4 (2 x (stub→kernel + kernel→stub))
- Similar concepts used in practice in DCOM and .NET

Asynchronous RPC

- RPC preserves the usual call behavior
  - The caller is suspended until the callee is done
- Potentially wastes client resources
  - Evident if if no return value is expected
  - In general, concurrency could be increased
- Many variants of asynchronous RPC (with different semantics):
  - If no result is needed:
    - Execution can resume after an acknowledgment is received from the server
    - One-way RPC returns immediately
      - May cause reliability issues: available in CORBA
      - "maybe semantics"
  - To deal with results, invocation may return immediately a *promise* (or *future*), later polled by the client to obtain the result

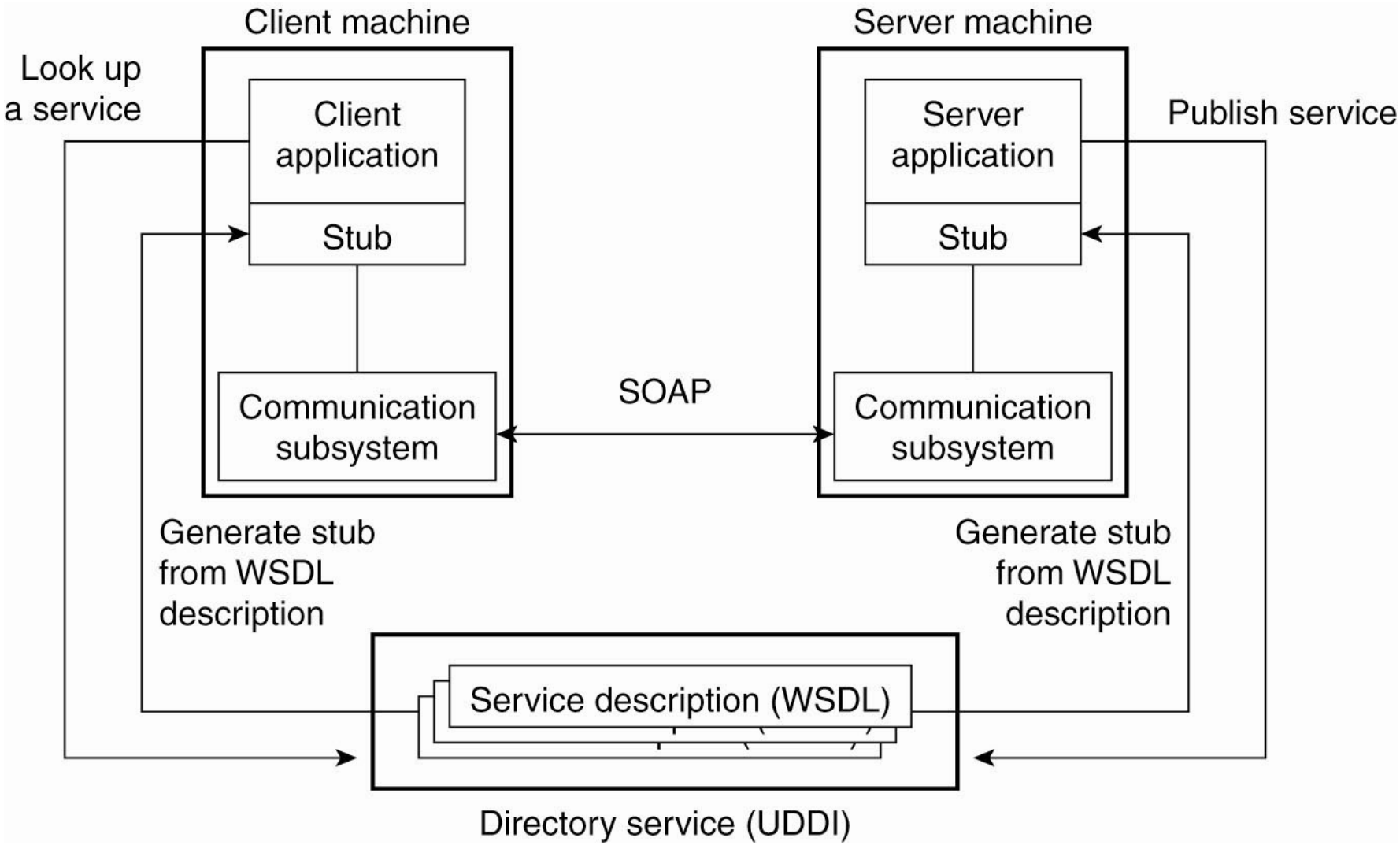


synchronous

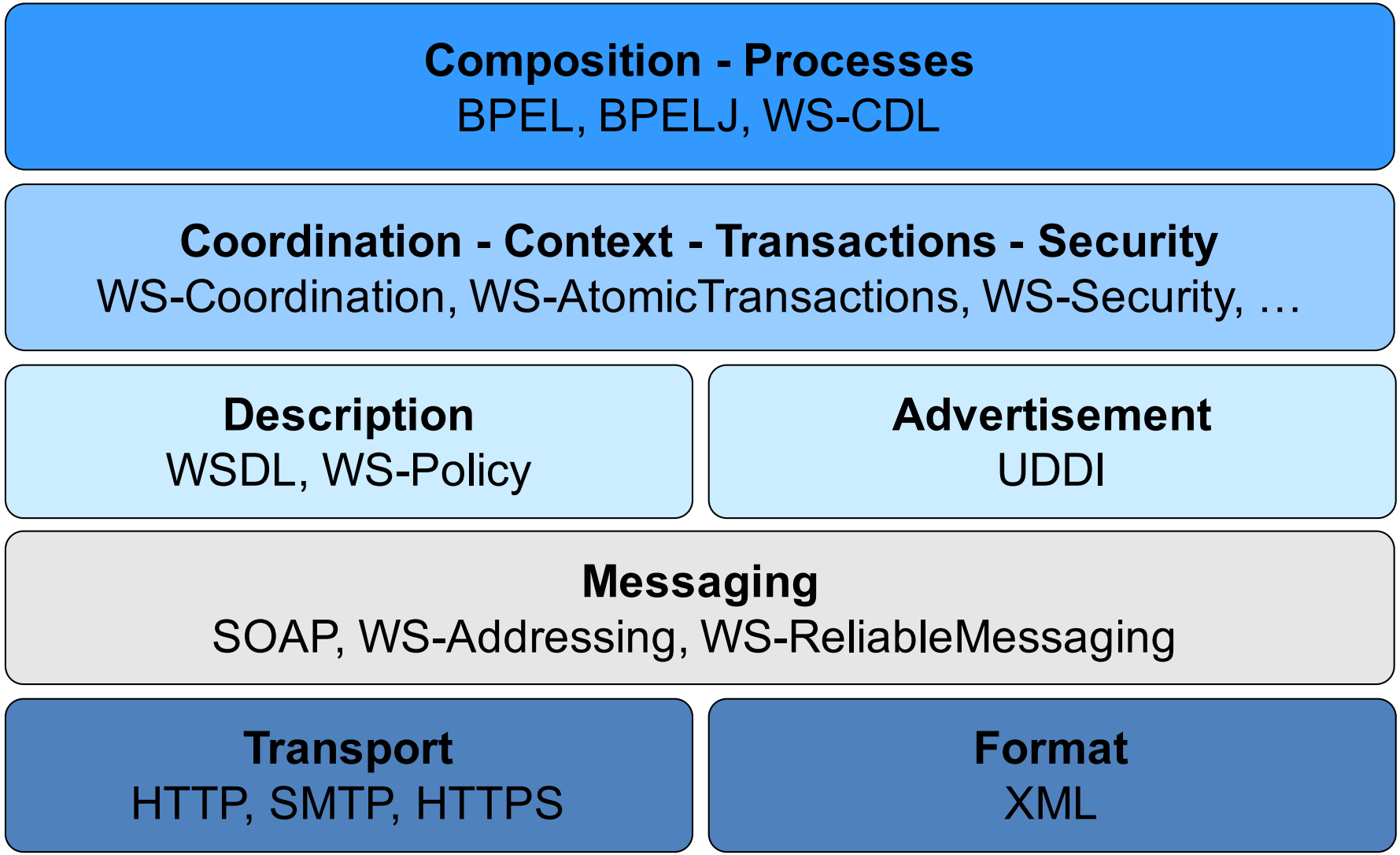


asynchronous
(one possible)

“Modern” RPCs: Web Services



“Modern” RPCs: Web Services



“Modern” RPCs: Web Services

- In a nutshell, they are services made available and accessible via common Internet/Web technology (e.g., HTTP, etc.)
  - At a high level of abstraction, they mimic the common structure of RPC systems
- The mechanics of the definition of and access to Web services is provided by a set of standards:
  - Services are specified in WSDL (Web Services Definition Language)
    - contains the precise definition of the interfaces of a service, e.g., signatures, data types, location, etc.
  - Services can be discovered via UDDI (Universal Description, Discovery, and Integration)
    - specifies how services are made available
    - plays the role of the directory service in RPC
  - The actual communication is based on SOAP (Simple Object Access Protocol)
    - defines the “wire format”, e.g., similar to XDR in ONC RPC

“Modern” RPCs: SOAP

- Roughly: RPC over HTTP
  - although support for SMTP (e-mail) is also defined
- The data format is based on XML (eXtensible Markup Language)
  - Initially created to extend HTML, has become a general means to describe structured data
  - human readable …
  - … but verbose! High communication and parsing costs

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Header>
      <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
        <n:priority>1</n:priority>
        <n:expires>2001-06-22T14:00:00-05:00</n:expires>
      </n:alertcontrol>
    </env:Header>
    <env:Body>
      <m:alert xmlns:m="http://example.org/alert">
        <m:msg>Pick up Mary at school at 2pm</m:msg>
      </m:alert>
    </env:Body>
</env:Envelope>
```

Serialization Frameworks

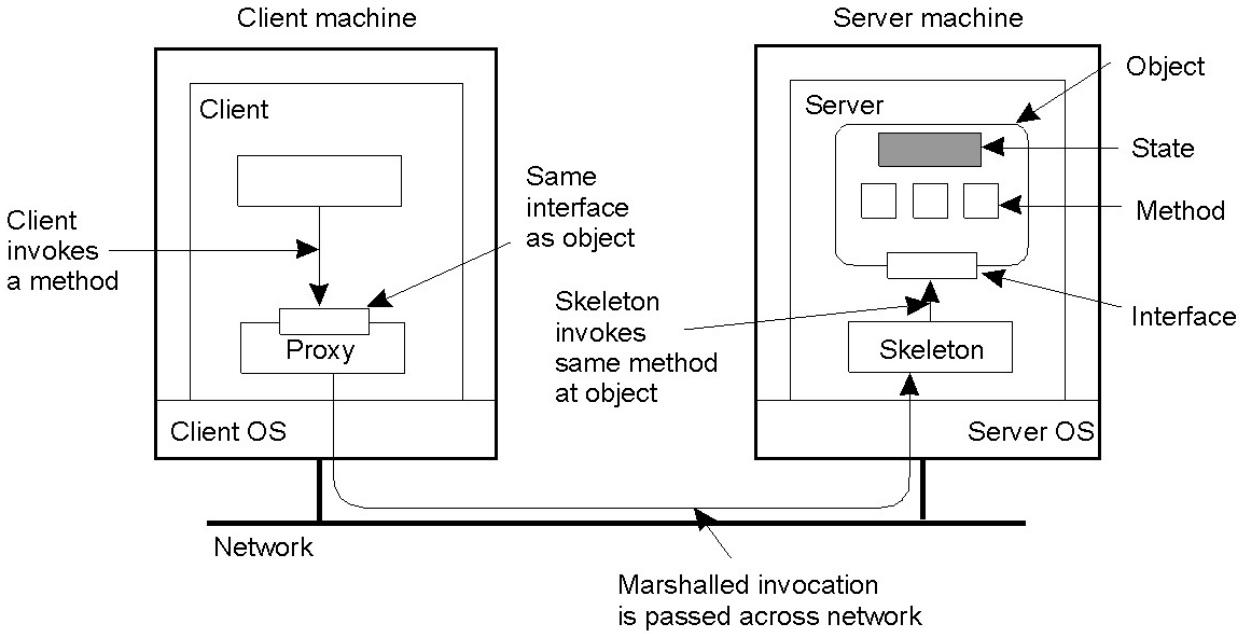
- Combination of an IDL and the wire format
- SOAP is known to be cumbersome/inefficient, so other frameworks were created
  - JSON (JavaScript Object Notation)
    - originally developed for JavaScript, today largely replaced SOAP
    - less powerful but much simpler, more efficient, widespread
  - Google Protocol Buffer
    - not human-readable, but much more efficient
    - used by Google's RPC (gRPC)
    - used internally by all Google backend services
  - Apache Thrifty
    - originally developed and used at Facebook
    - a "better" Google Protocol Buffer
  - Apache Avro
    - the most recent
    - data is always accompanied by a "schema" that supports the processing/analysis of the data; this notably
      - removes the need for code generation (IDL compilation)
      - reduces the amount of data to be serialized (no tags)

Distributed Object Middleware

- Same idea as RPC, different programming constructs
  - The aim is to obtain the advantages of OOP also in the distributed setting
- Important difference: remote object references can be passed around
  - Need to maintain the aliasing relationship
- Shares many of the core concepts and mechanisms with RPC
  - Sometimes built on top of an RPC layer

Interface Definition Language

- In RPC, the IDL separates the interface from the implementation
  - To handle platform/language heterogeneity
- Such separation is one of the basic OO principles
  - It becomes natural to place the object interface on one host, and the implementation on another
- The IDLs for distributed objects are much richer
  - Inheritance, exception handling, …



Java Remote Method Invocation

- Java RMI focuses only on remote method invocation
  - Kind of RPC for objects…
  - More advanced services provided by other components of the Java family
- Part of Java since version 1.1
- Unlike RPC, no need to use an IDL compiler
  - Java interfaces serve as an IDL
  - The Java Serialization framework defines the wire format
    - and can be redefined by the programmer
  - dynamic proxies: automatically generated classes implementing a set of interfaces specified at run-time

Interfaces and Remote Objects

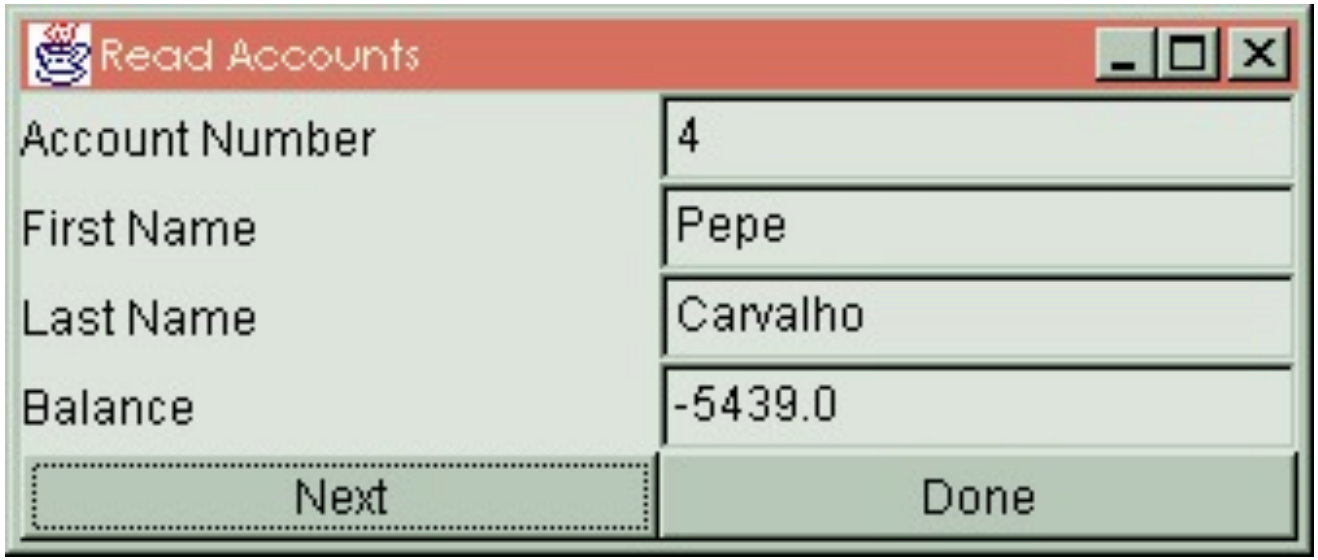
- The Java interface concept acquires a new role with distribution
  - Used as an IDL specification, without an IDL compiler
  - possible because RMI is designed to support only Java
- A ***remote object*** (i.e., one whose methods can be invoked remotely) must implement an interface that extends **`java.rmi.Remote`**
  - And whose methods must throw **`java.rmi.RemoteException`**
- Objects can also be copied around using serialization, as long as their class implements **`java.io.Serializable`**

Obtaining a Remote Reference

- There are essentially two ways:
  1. Through parameter passing, as an input parameter or result value (just like in Java)
  2. By expliciting querying a simple directory services called **`rmiregistry`**
- In both cases, the remote reference is an object containing the client stub (proxy)
  - Implements the same interface as the remote object
  - Instance of **`RemoteStub`**
  - Proxies are serializable (i.e., can be passed around)
  - Possible because there is only one language …
- Once acquired the remote reference is indistinguishable from a local one
  - the client can invoke any of the methods in the remote interface of the target object

Example: Remote Access to Accounts

- The server enables access to the content of the local file **`accounts.dat`**
- Remote clients can access the server and browse (sequentially) the accounts



Example: Server

```
public interface AccountServer extends java.rmi.Remote {
  Account getAccount(int number) throws RemoteException;
}

public class AccountServerImpl
 extends java.rmi.server.UnicastRemoteObject
 implements AccountServer {
 private Hashtable accounts;
 …
 public static void main(String[] args) {
  …
  try {
   Naming.rebind("//localhost:"+ String.valueOf(port) +
                 "/AccountServer", new AccountServerImpl());
 } catch(java.net.MalformedURLException mue) { … }
 } catch(RemoteException re) { … }
 }
 public AccountServerImpl() throws RemoteException { … }

 public Account getAccount(int number)
        throws RemoteException {
    return (Account) accounts.get(new Integer(number));
 }
}
```

A Server Object's Life

- Create an instance of a remote object
- ***Export*** it so that it can receive and respond to remote calls
- Make it known to potential clients, by either
  - publishing it in a lookup service (the **`rmiregistry`**)
  - passing it as a parameter (input or output) in remote method invocations on other remote objects

Exporting Remote Objects

- Implementing a remote interface is not sufficient: to accept remote calls a remote object must be ***exported***
  - Basically, to listen on a socket for incoming calls
  - Done automatically in the constructor, if the remote object subclasses from **`java.rmi.server.RemoteObject`**
    - Typically, the **`UnicastRemoteObject`** class is used
- Alternately, done with the static method **`UnicastRemoteObject.exportObject(Remote)`**
  - E.g., when there is a need to inherit from an application class
  - Returns the stub
- In any case, all the constructors of the remote object must throw a **`RemoteException`**
- The object can be unexported, too

Example: Client

```
…
try {
 accountServer =
   (AccountServer) Naming.lookup("//" + host +
   ":" + String.valueOf(port) +
   "/AccountServer");
} catch(NotBoundException nbe) { … }
} catch(RemoteException re) { … }
} catch(java.net.MalformedURLException mue) { …
   }
}
…
Account a = null;
try {
 a = accountServer.getAccount(current++);
} catch (RemoteException re) { … }
…
```

Apart from the try/catch block, the remote method invocation looks just like a local one

A Client Object's Life

- Clients must obtain a reference to the remote object
  - From the registry or as part of a remote method invocation
- Methods can then be invoked as usual on the remote object, catching the appropriate exceptions
- The same remote object can be shared among different clients
  - Their requests can be serialized or processed in parallel
  - The specification makes no guarantee as to which approach is taken

TCP Sockets in Java

- Connections on the client side are handled through a **`Socket`** class
  - An instance is created by passing host and port to the constructor
  - Alternatively, an unconnected socket can be created with the parameterless constructor, and later connected using **`connect`**
  - Methods are provided to get information about the socket, set network parameters (e.g., timeout, buffer size), …
- The server side is managed by a **`ServerSocket`**
  - Must be bound to a port
    - done either in the constructor or using the **`bind`** method
  - Listening begins when **`accept()`** is called
    - Blocking call: the thread is suspended until a client connects
    - **`accept`** returns the socket instance representing the client connection
- A **`close()`** method is provided on both classes
- Java also supports UDP sockets

Handling Communication

- Communication occurs through I/O streams, which are attached to the **`Socket`** instance
- The streams are obtained with **`getInputStream`** and **`getOutputStream`**
  - The result must be "decorated" with appropriate I/O classes
    - e.g., **`DataInputStream`**, **`ObjectInputStream`**, **`GZIPInputStream`**, …
- After the streams are obtained, it is just I/O
- However, the point is that client and server must agree on the format of messages and their semantics
  - I.e., they must agree on an ***application-level protocol***

Example: The Client

```
…
InetAddress host = …
s = new Socket(host, port);
input = new DataInputStream(s.getInputStream());
output = new DataOutputStream(s.getOutputStream());
…
public void readRecord() {
  try {
    output.writeShort(NEXT);
    accountNumber = input.readInt()
    first = input.readUTF();
    last = input.readUTF();
    balance = input.readDouble();
  } catch (EOFException e) { closeConnection();
  } catch (IOException e) { System.exit(1); }
}
```

Data serialization is explicitly handled by the programmer

Java also offers automated (and customizable) serialization but this is not necessarily the case for all languages supporting sockets!

Example: The Server

```
try {
  server = new ServerSocket(port);
  while(true) {
  client = server.accept();
  is = new DataInputStream(client.getInputStream());
  os = new DataOutputStream(client.getOutputStream());
  DataInputStream dataStream =
    new DataInputStream(new FileInputStream(DATAFILE));
  int command = -1;
  while(command != DONE) {
    command = is.readShort();
    switch(command) {
      case NEXT:
      os.writeInt(dataStream.readInt());
      os.writeUTF(dataStream.readUTF());
      os.writeUTF(dataStream.readUTF());
      os.writeDouble(dataStream.readDouble());
      break;
      case DONE:
      dataStream.close();
      is.close();os.flush();os.close();client.close();
      break;
    }
  }
} catch(IOException ioe) { ioe.printStackTrace(); }
```

The application-level protocol is described in a very low-level, error-prone way

Serving Multiple Clients

- The server cannot accept new connections unless it blocked on an **`accept()`**
  - In the example, this occurs only when the client terminates the connection, and the server goes back to listening on the socket
- Two solutions:
  - Using multithreading:
    - A thread listens on the server socket, and dispatches the client connection to another thread
    - Typically, one thread per client
  - Using a different I/O library (>1.4)
    - Multiple connections are multiplexed behind a non-blocking interface
    - A single thread can poll multiple sockets

More on RMI: Parameter Passing

- The semantics of invocation is different in the local and in the remote case
  - As a result of a precise design choice!
- Given a method invocation `m(obj)`:
  - If `obj` is a remote object, the usual by-reference semantics is preserved: if `m` modifies the state of `obj`, the latter is accessed directly through the network
  - Otherwise, `obj` is passed by copy; modifications in `m` are visible only on the copy and do not trigger network communication
- The same holds for methods with multiple input parameters, as well as for result values
- Trades ease of programming for communication efficiency
- Still unsatisfactory: the passing modality can be decided (to a large extent) only statically, and not per-invocation or per-instance

More on RMI: Code Downloading

- ***Problem:*** Consider an invocation `m(T a)`
  - We can expect that the bytecode for `T` is co-located with the remote object
  - But, due to polymorphism, the client may send a subtype of `T`
    - Always the case when `T` is an interface
  - Same problem regardless of whether `a` is a remote object or not
    - Finding the stub class vs. the actual class
- RMI is designed as an open system: cannot assume all possible types are preloaded everywhere
- ***Solution:*** applications can annotate types with codebase information
  - i.e., specify where to find the bytecode for a given type name
  - typically a Web server
- Implemented by redefining the ***class loader*** and the serialization mechanism
- Simplifies deployment of stubs

More on RMI: Distributed Garbage Collection

- Garbage collection across the network is performed by coordinating the JVMs of client and server objects
  - When a client obtains a remote reference, its JVM calls **`dirty()`** on the JVM of the server object
  - Dually, when the client no longer needs the remote reference, its JVM calls **`clean()`**, notifying the server JVM
- Remote references are temporary, and must be renewed via leases
  - A call to **`dirty()`** returns a lease for a given (configurable) time
  - The lease must be refreshed with another **`dirty()`** to keep the remote reference alive
  - Network problems may prevent the client from refreshing the lease
- A server object in the **`rmiregistry`** exports a local reference to the latter, and therefore is not garbage collected
- The **`Unreferenced`** interface, if implemented, allows a server to perform cleanup operations when clients are no longer accessing it

Common Object Request Broker Architecture (CORBA)

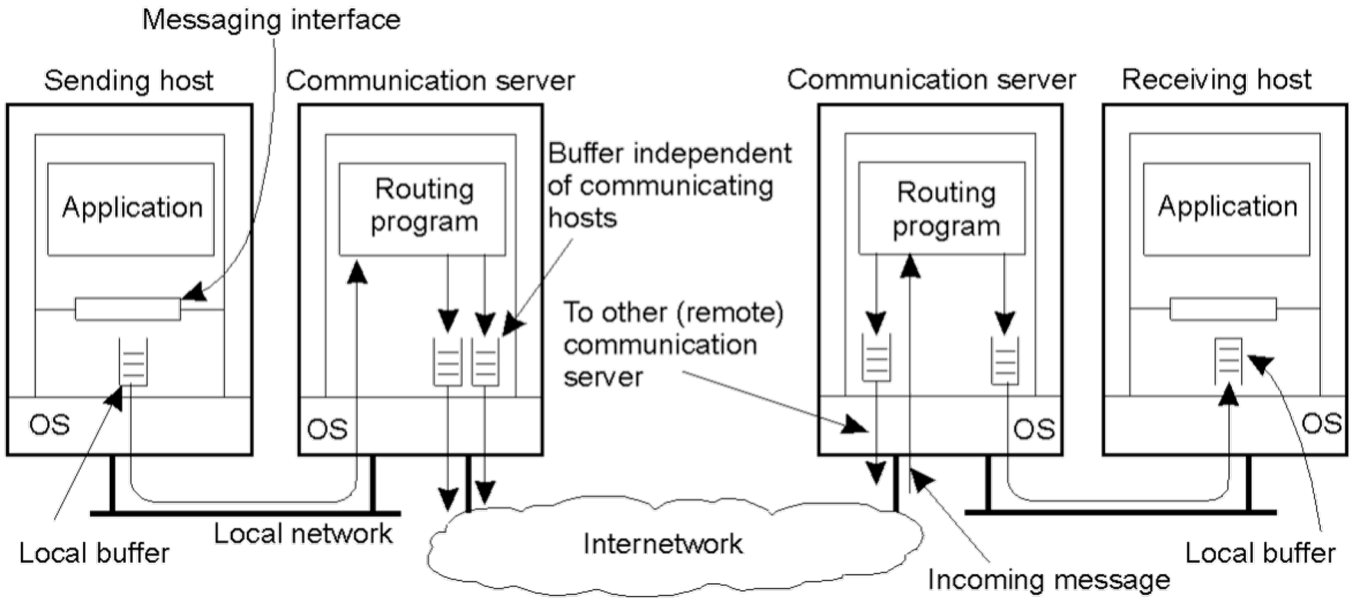
- CORBA is specified by the Object Management Group (OMG) a non-profit organization with over 800 members founded in 1989
  - The same responsible of the standardization of UML
  - Specifications and more info at **`www.corba.org`**
- Main focus: interoperability
- Huge standard, plenty of features
  - Implementations provide their own extensions
- First specifications became available in early 90s
  - Object Management Architecture (OMA) contains the reference architecture and was specified in 1992; CORBA in 1995
  - CORBA 3.3 is the latest version released (in 2012)
  - Free implementations available
  - A limited CORBA implementation ships with J2SE
- Today: still used but mostly in niche/legacy domains

Message-Oriented Middleware

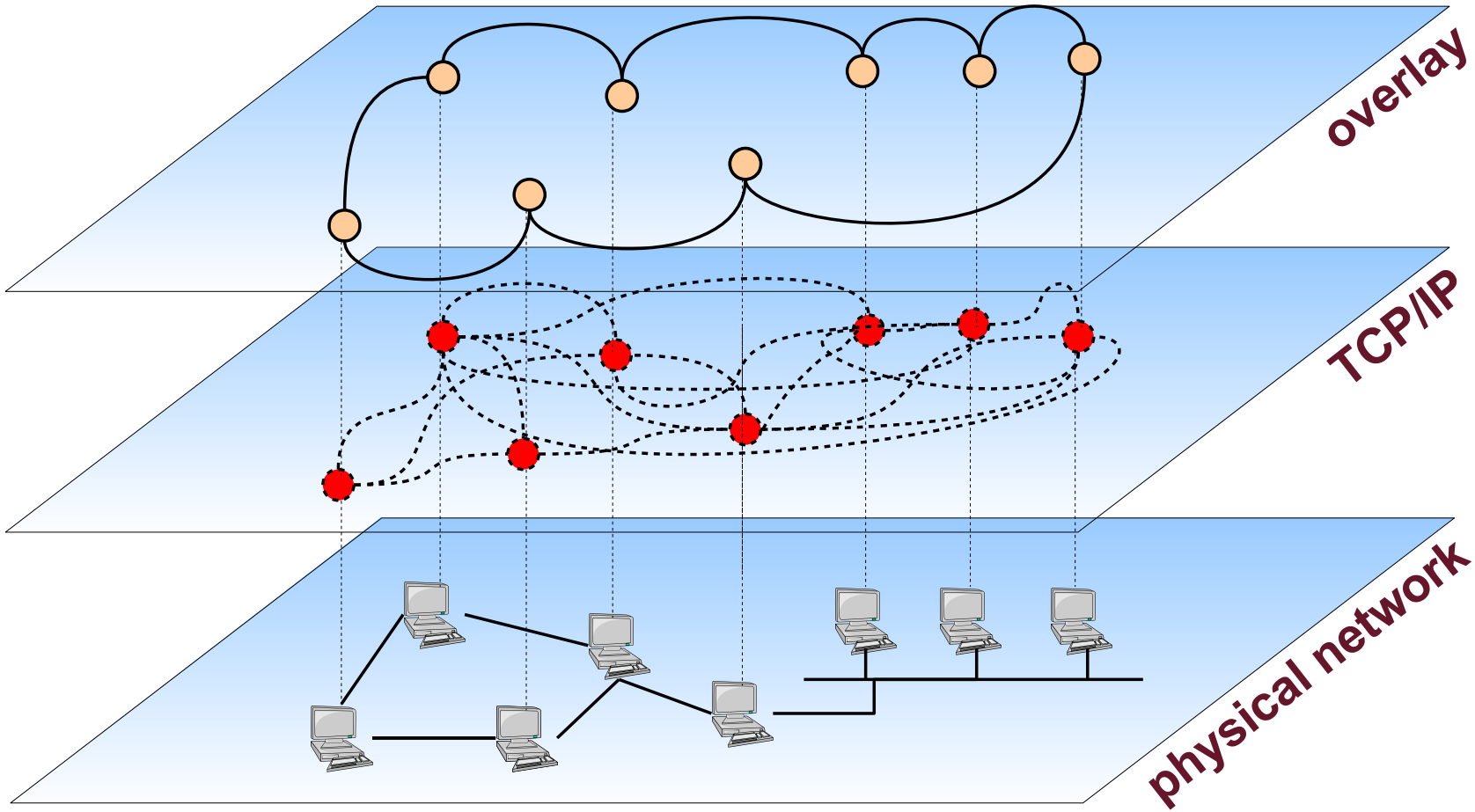
- Remote procedure/method call foster a synchronous model
  - Inspired by routine invocation
- Natural programming abstraction, but:
  - Supports only point-to-point interaction
  - Synchronous communication is expensive
  - Intrinsically tight coupling between caller and callee
    - “Session oriented”, leads to “rigid” architectures
- MOMs are asynchronous communication models
  - Often centered around the notion of message/event
    - Or persistent communication
  - Often supporting multi-point interaction
  - Bring more decoupling among components

Reference Model

- The most straightforward form of asynchronous communication is message passing
  - Typically directly mapped on/provided by the underlying network OS functionality (e.g., socket, datagrams)
  - Quite low-level
- The messaging infrastructure is often provided at the application level, by several “communication servers”
  - Through what is nowadays called an ***overlay network***



Overlay Networks



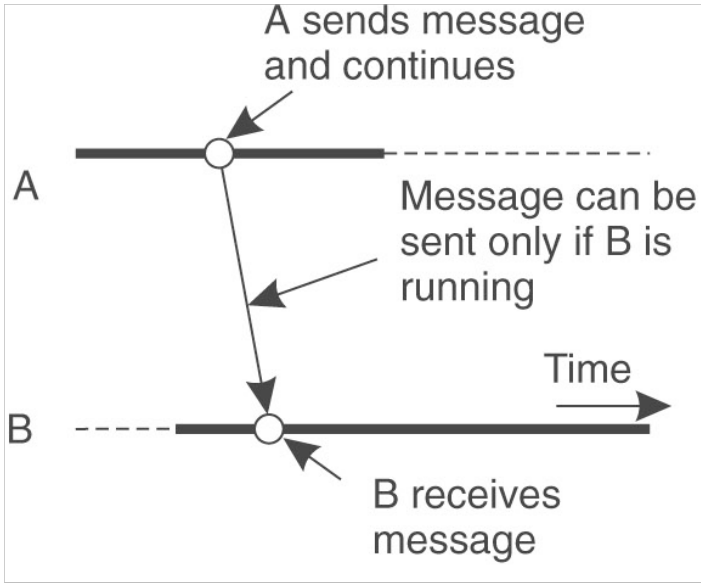
- Tremendous design flexibility in providing application-specific topologies and protocols

Types of Communication

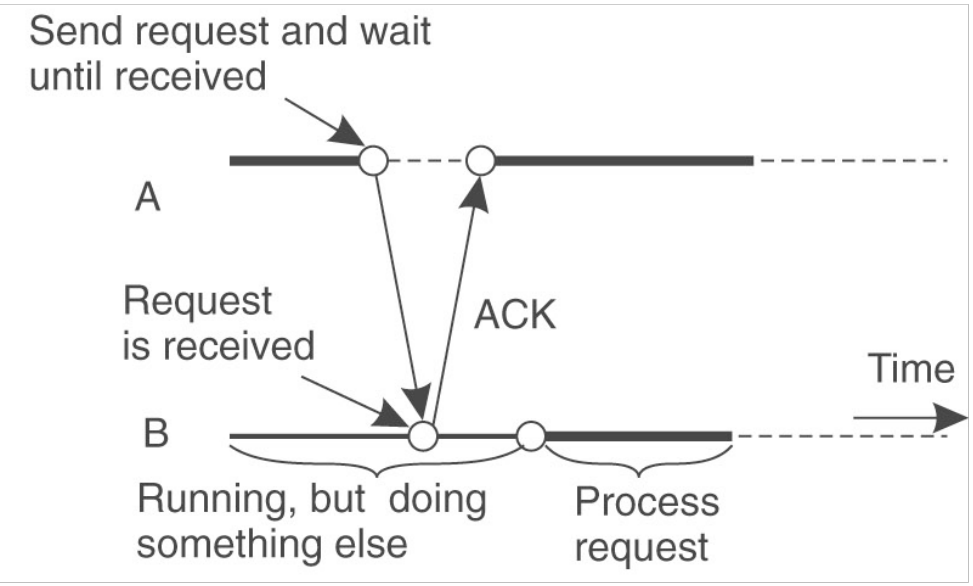
- Synchronous vs. asynchronous
  - ***Synchronous***: the sender is blocked until the recipient has stored (or received, or processed) the message
  - ***Asynchronous***: the sender continues immediately after sending the message
- Transient vs. persistent
  - ***Transient***: sender and receiver must both be running for the message to be delivered
  - ***Persistent***: the message is stored in the communication system until it can be delivered
- Several alternatives (and combinations) are provided in practice
- Many are used also for implementing synchronous models (e.g., RPC)

Transient Communication

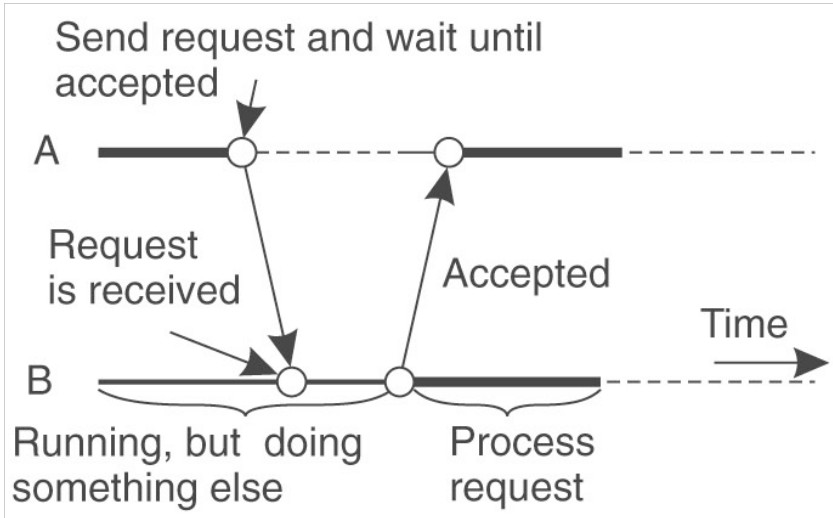
Asynchronous



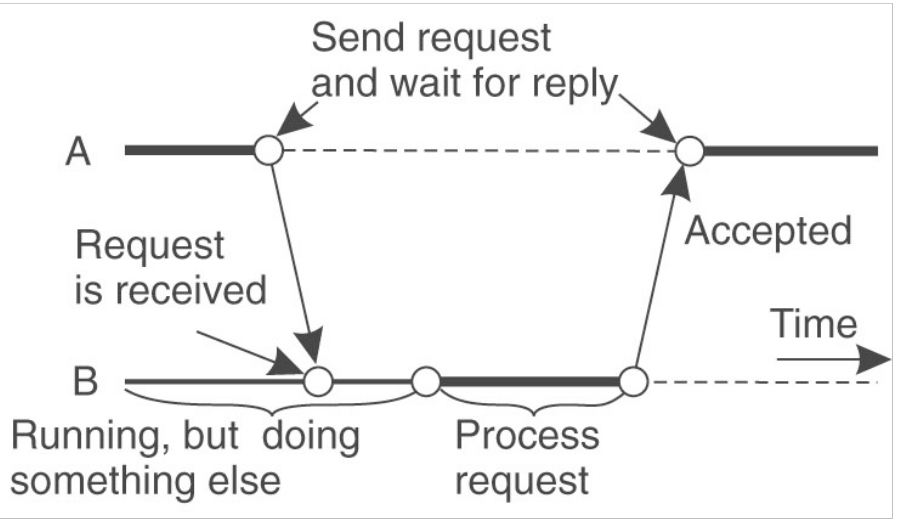
Receipt-based synchronous



Delivery-based synchronous

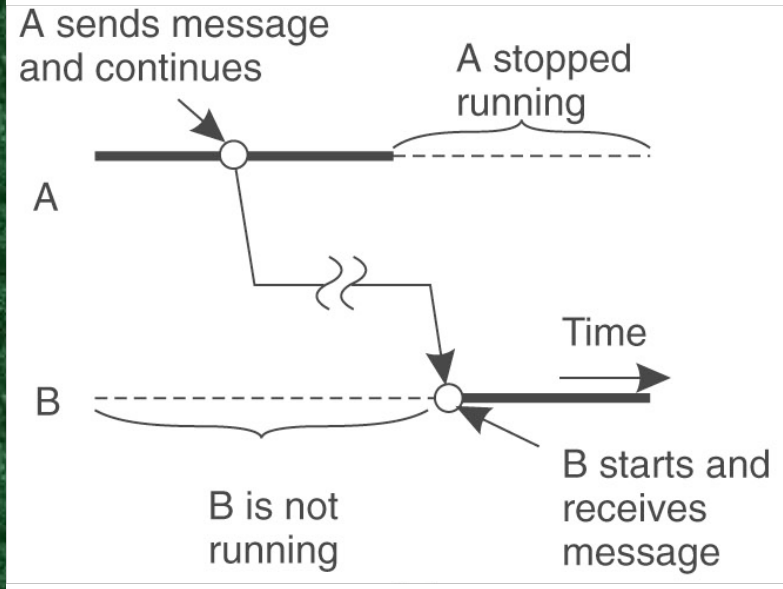


Response-based synchronous

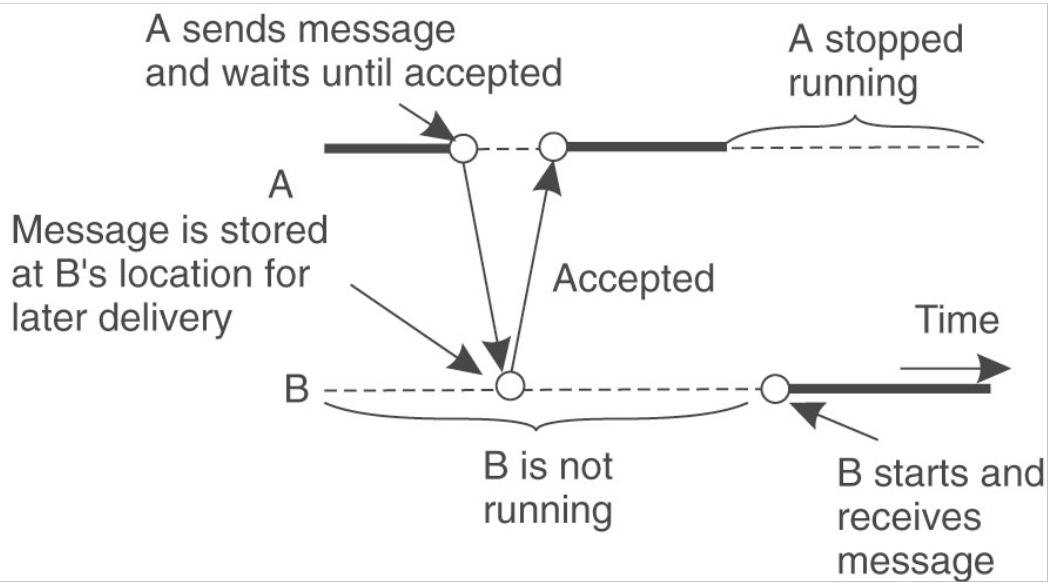


Persistent Communication

Asynchronous



Synchronous



Different MOM Flavors…

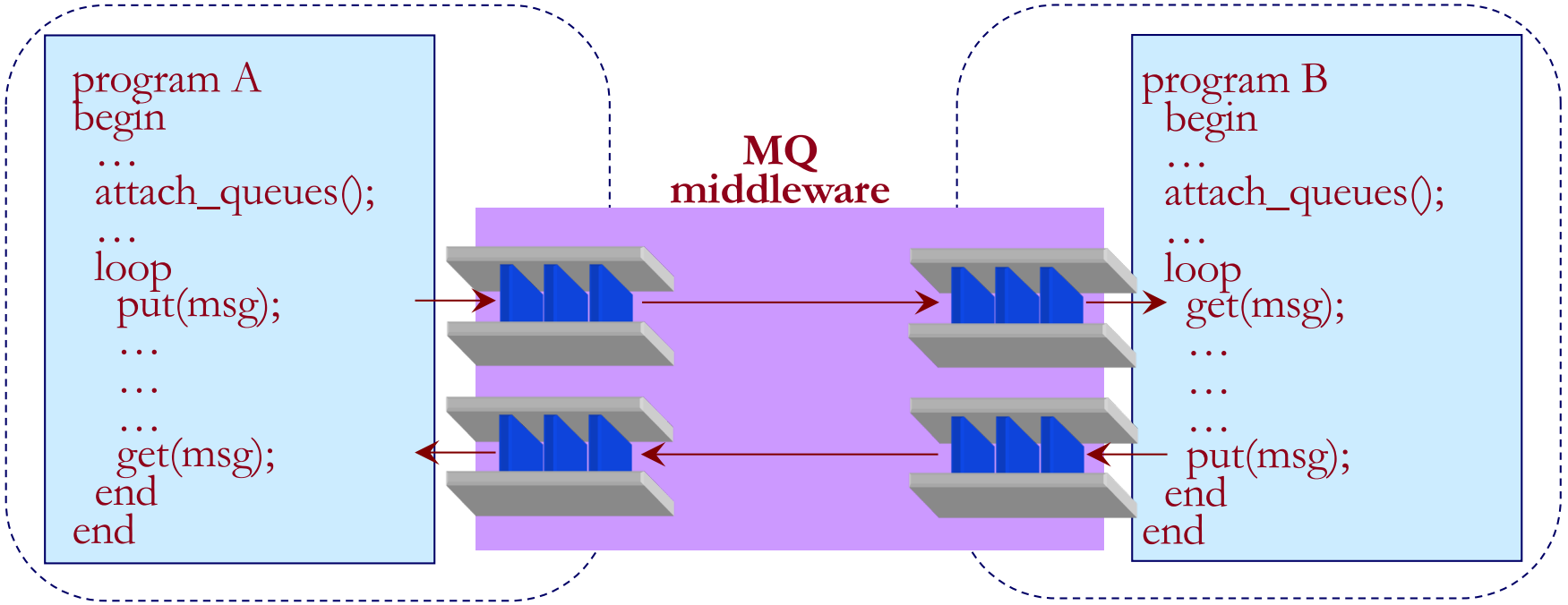
- There are two main classes of MOMs:
  - message queuing
  - publish-subscribe
- Several common characteristics, to the point that often the distinction is blurred
  - Both are “message oriented”
  - Both offer a strong decoupling among components
  - Both are often based on a set of “servers” to route messages from sender to recipients and/or to support persistency
- But also several differences...

Message Queuing

- Point-to-point persistent asynchronous communication (and others)
  - Typically guarantee only eventual insertion of the message in the recipient queue (no guarantee about the recipient's behavior)
  - Communication is decoupled in time and space
  - Can be regarded as a generalization of e-mail
- Originally geared towards:
  - Integration of large-scale enterprise systems
  - Long message transfers (minutes)
- Each component holds an input queue and an output queue
- Intrinsically peer-to-peer architecture
- Many commercial systems:
  - IBM MQSeries (now WebSphere MQ), DECmessageQ, Microsoft Message Queues (MSMQ), Tivoli, Java Messaging Service (JMS), …

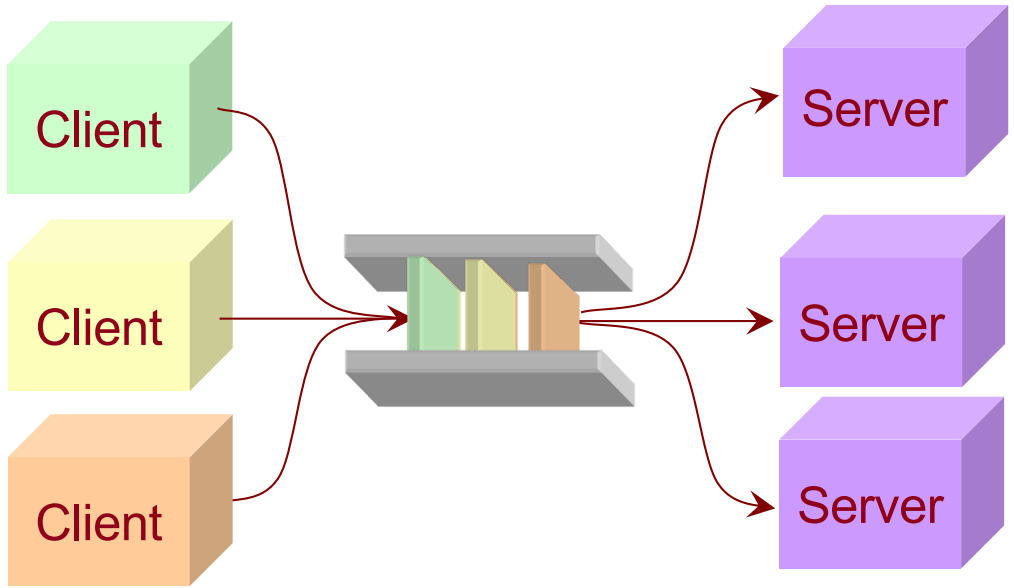
Communication Primitives

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue



Client-Server with MQ

- Clients send requests to the server's queue
- The server asynchronously fetches requests, processes them, and returns results in the clients' queues
- Thanks to persistency and asynchronicity, clients need not remain connected
- Queue sharing simplifies load balancing



Message Queuing Services

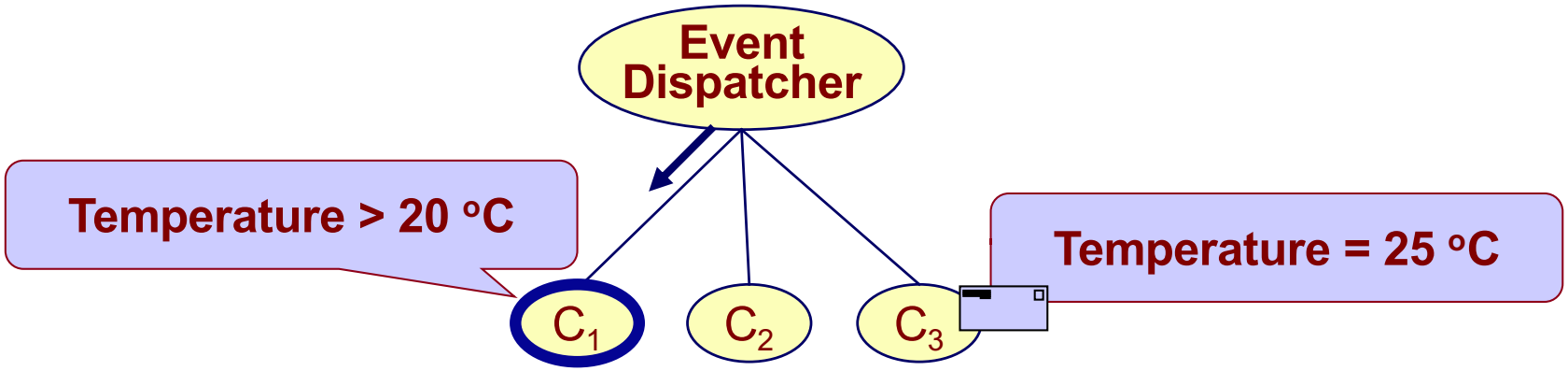
- Queue creation and removal
  - Persistent vs. transient
  - Queue groups
  - Support for remote queues
- Message transport
  - Point-to-point vs multi-point
  - Priorities
  - Several variants for delivery guarantees
- Message receipt
  - Queues can be shared among components
  - FIFO receipt is default, but several options are provided
    - E.g., based on priority, user-defined filters, …
- Directory services, security, transactional services, …

Architectural Issues

- Queues are identified by symbolic names
  - Need for a lookup service, possibly distributed
  - Often pre-deployed static topology
- Queues are manipulated by queue managers
  - Local and/or remote, acting as ***relays*** (aka applicative routers)
- Relays often organized in an overlay network
  - Messages are routed by using application-level criteria, and by relying on a partial knowledge of the network
  - Improves fault tolerance
  - Provides applications with multi-point without IP-level multicast
- ***Message brokers*** provide application-level gateways supporting message conversion
  - Useful when integrating sub-systems

Publish-Subscribe

- Application components can publish asynchronous ***event notifications***, and/or declare their interest in event classes by issuing a ***subscription***
  - Extremely simple API: only two primitives (**`publish`**, **`subscribe`**)
  - Event notifications are simply messages
- Subscriptions are collected by an ***event dispatcher*** component, responsible for routing events to all matching subscribers
  - Can be centralized or distributed
- Communication is transiently asynchronous, implicit, multipoint
- High degree of decoupling among components
  ➔ easy to add and remove components
  ➔ appropriate for dynamic environments

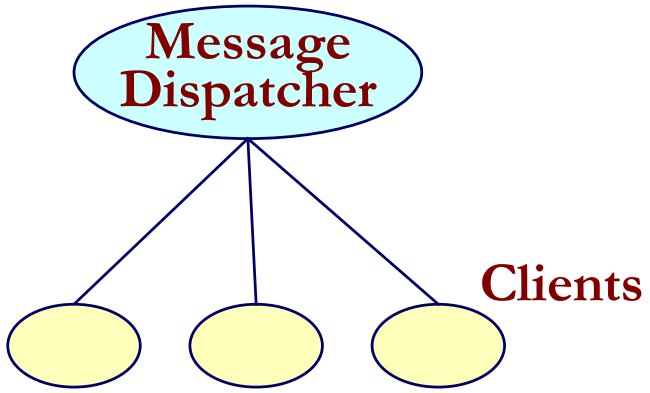


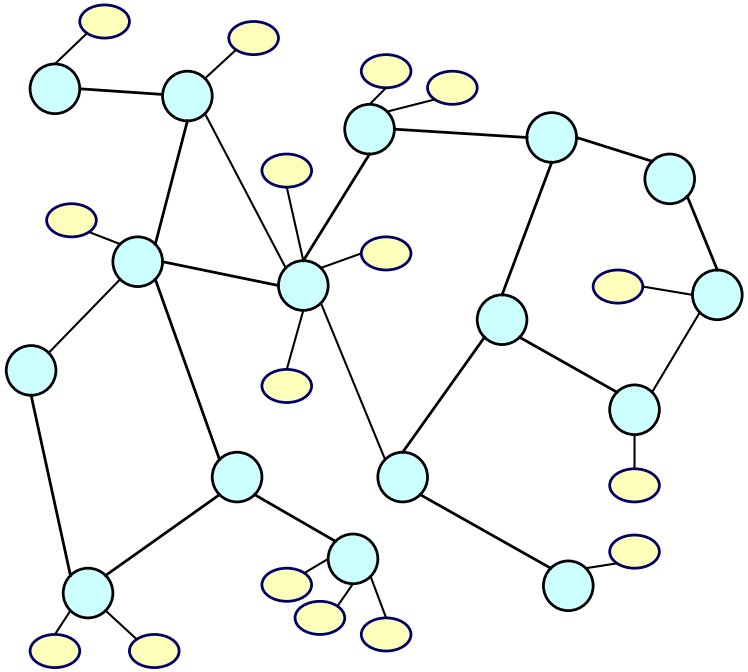
Subscription Language

- The expressiveness of the subscription language allows one to distinguish between:
  - ***Subject-based (or topic-based)***
    - The set of subjects is determined a priori
    - Analogous to multicast
    - Example: subscribe to all events about “Distributed Systems”
  - ***Content-based***
    - Subscriptions contain expressions (***event patterns***) that allow clients to filter events based on their content
    - The set of patterns is determined by client subscriptions
    - A single event may match multiple subscriptions
    - Example: subscribe to all events about a “Distributed System” class with date greater than 16.11.2004 and held in classroom D04
- The two can be combined
- Tradeoffs:
  - complexity of the implementation vs. expressiveness
  - However, expressiveness allows additional filtering!

Architecture of the Dispatcher

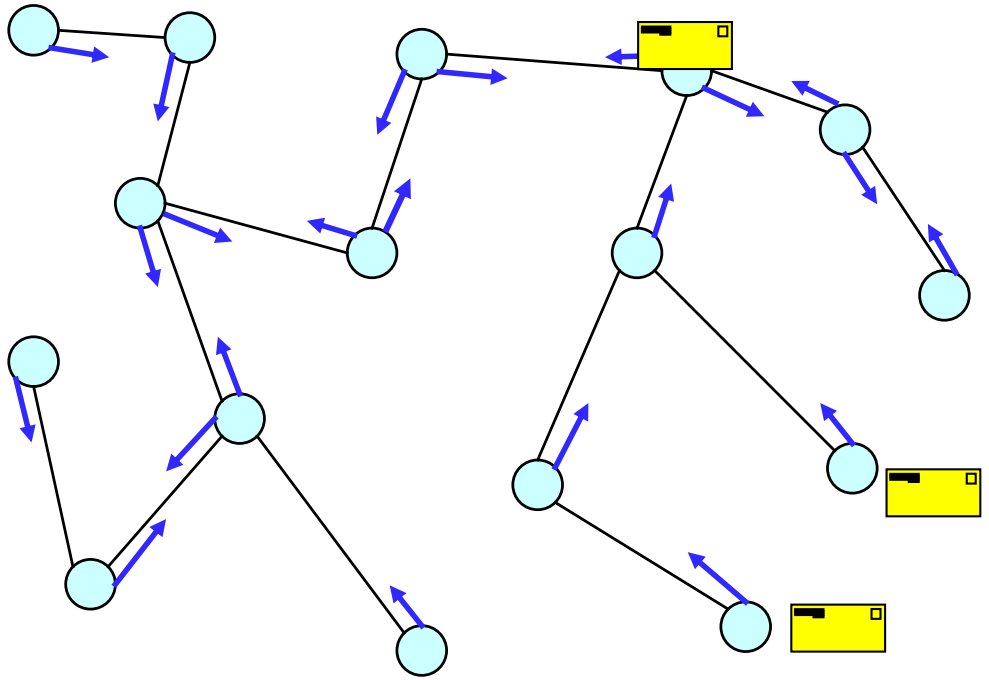
- Centralized
  - A single component is in charge of collecting subscriptions and forward messages to subscribers
- Distributed
  - A set of ***message brokers*** organized in an ***overlay network*** cooperate to collect subscriptions and route messages
  - The topology of the overlay network and the routing strategy adopted may vary





Subscription Forwarding

- Each dispatcher forwards subscriptions to neighbors
  - Subscriptions are never sent twice over the same link
- Events follow the routes laid by subscriptions
- Optimizations may exploit coverage relationships
  - E.g., "**`Distributed *`**" > "**`Distributed systems`**"
  - Fusion, subsumption, summarization



So What?

- Several kinds of middleware, different features, performance, strengths and weaknesses
  - Some middleware platforms tend to integrate different paradigms in a single system, but features are not independent
- The ultimate middleware does not exist
  - Much like the ultimate programming language
- Different applications require different abstractions
  - Possibly more than one in the same application
- The most appropriate middleware must be evaluated not only in terms of performance and efficiency, but also expressiveness, flexibility, communication paradigm