Module 3 - Distributed Objects & Remote Invocation

Programming Models for Distributed Applications

- Remote Procedure Call (RPC)
  - Extension of the conventional procedure call model
  - Allows client programs to call procedures in server programs
- Remote Method Invocation (RMI)
  - Object-oriented
  - Extension of local method invocation
  - Allows an object in one process to invoke the methods of an object in another process
- Message-Based Communication
  - Allows sender and receiver to communicate without blocking and without the receiver having to be running

Essentials of Interprocess Communication

- Ability to communicate - exchange messages
  - This is the responsibility of the networks and the request-reply protocol
- Ability to "talk" meaningfully
  - Interfaces
  - Processes have to be able to understand what each other is sending
    - → Agreed standards for data representation
    - → We'll discuss this later

Interface

- Defines what each module can do in a distributed system
- Service interface (in RPC model)
  - specification of procedures offered by a server
- Remote interface (in RMI model)
  - specification of methods of an object that can be invoked by objects in other processes
- Example: CORBA IDL

```
//In file Person.idl
Struct Person {
   string name;
   string place;
   long year;
};
Inerface PersonList {
   readonly attribute string listname;
   void addPerson(in Person p);
   void getPerson(in string name, out Person p);
   long number();
};
```

Remote Procedure Call (RPC)

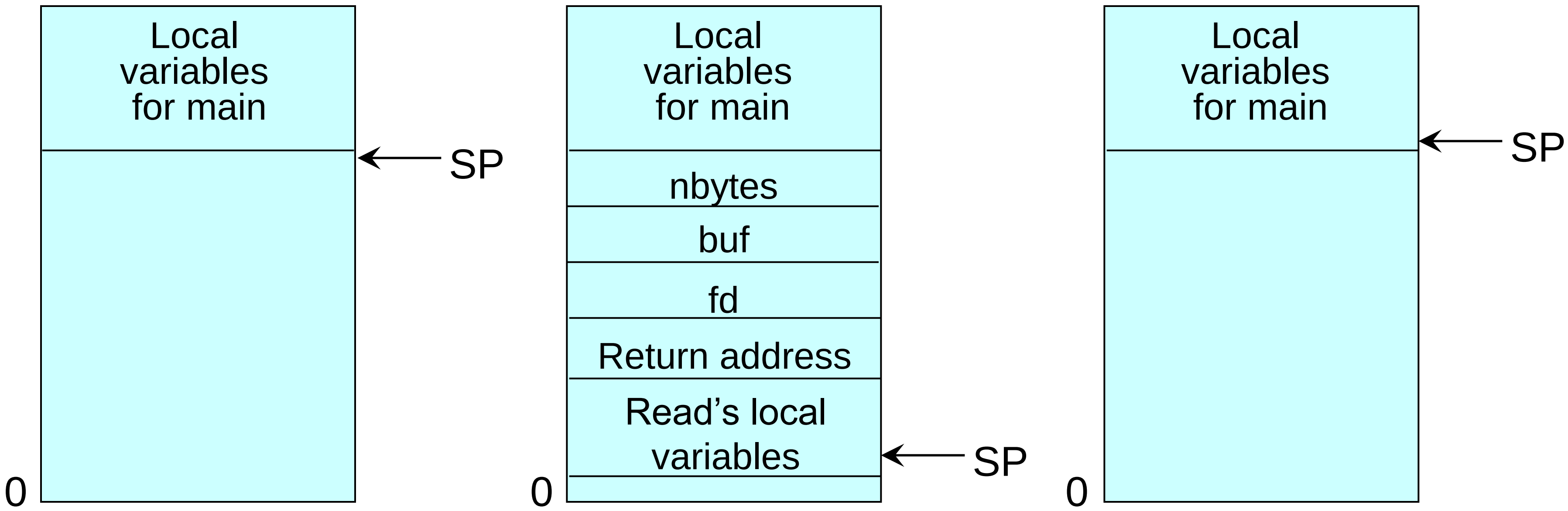
- Extension of the familiar procedure call semantics to distributed systems (with a few twists)
- Allows programs to call procedures on other machines
- Process on machine A calls a procedure on machine B:
  - The calling process on A is suspended; and execution of the called procedure takes place on B;
  - Information can be transported from the caller to the callee in the parameters; and information can come back in the procedure result
  - No message passing or I/O at all is visible to the programmer
- Subtle problems:
  - Calling and called procedures execute in different address spaces
  - Parameters and results have to be passed, sometimes between different machines
  - Both machines can crash

Local Procedure Call

- Consider a C program with the following call in the main program

```
count = read(fd, buf, nbytes)
```

where **fd** (file descriptor) and **nbytes** (no. bytes to read) are integers, and **buf** (buffer into which data are read) is an array of characters



Before call to read

While read is active

After return to main

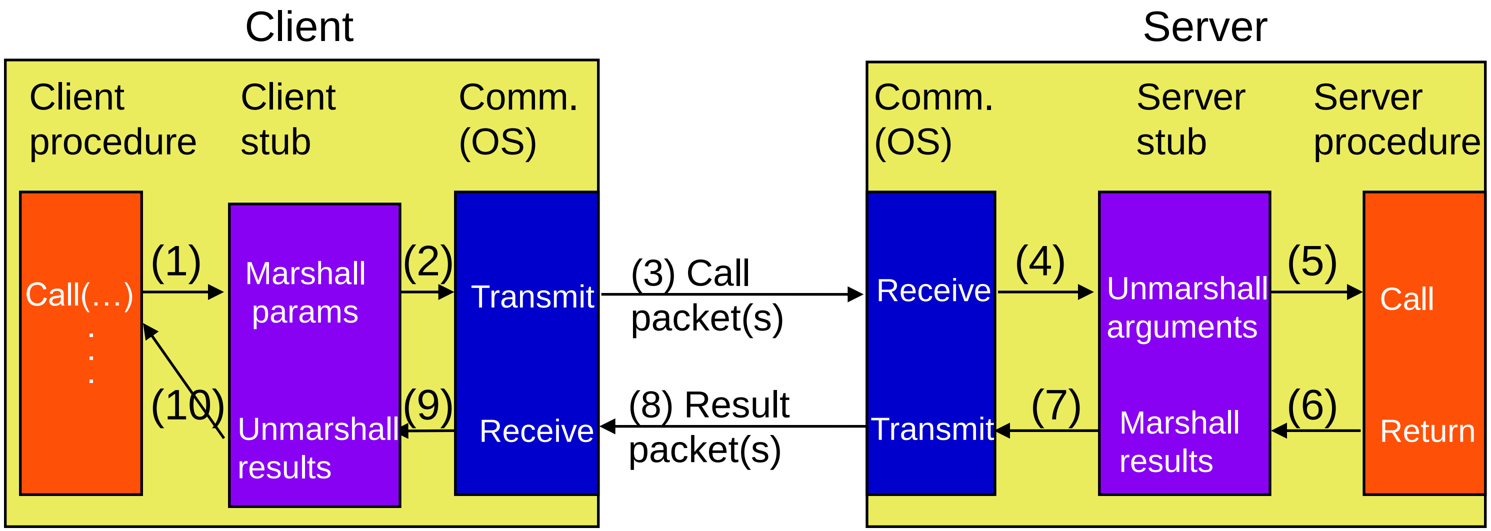
The read routine is extracted from library by linker and inserted into the object program. It puts the parameters in registers and issues a READ system call by trapping to the kernel.

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

RPC Operation

- Objective: Make interprocess communication among remote processes similar to local ones (i.e., make distribution of processes transparent)
- RPC achieves its transparency in a way analogous to the read operation in a traditional single processor system
  - When read is a remote procedure (that will run on the file server's machine), a different version of read, called client stub, is put into the library
  - Like the original read, client stub is called using the calling sequence described in previous slide
  - Also, like the original read, client stub traps to the kernel
  - Only unlike the original read, it does not put the parameters in registers and ask the kernel to give it data
  - Instead, it packs the parameters into a message and asks the kernel to send the message to the server
  - Following the call to send, client stub calls receive, blocking itself until the reply comes back

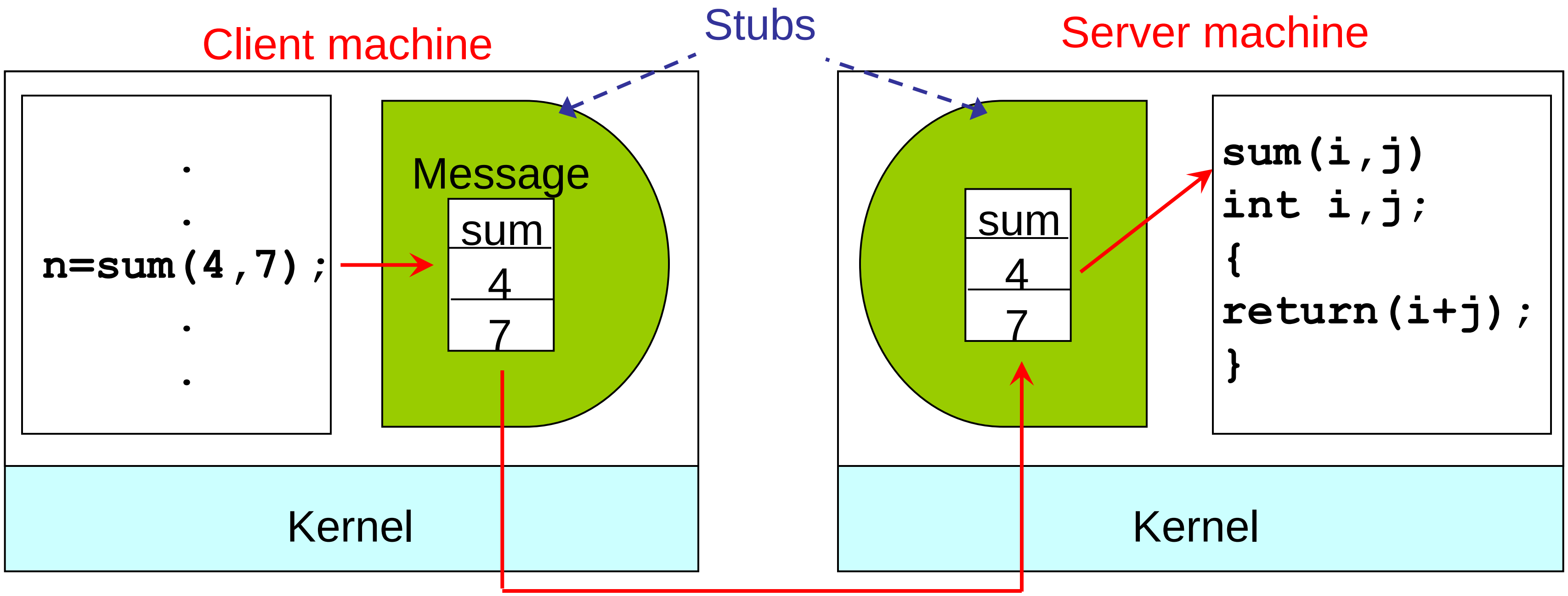
RPC Operation (2)



1. Client procedure calls the stub as local call
2. Client stub builds the message and calls the local OS
3. Client OS sends msg to server OS
4. Server OS hands over the message to server stub
5. Server stubs unpacks parameters and calls server procedure
6. Server procedure does its work and returns results to server stub
7. Server stub packs results in a message and calls its local OS
8. Server OS sends msg to client OS
9. Client OS hands over the message to client stub
10. Client stubs unpacks result and returns from procedure call

Parameter Passing

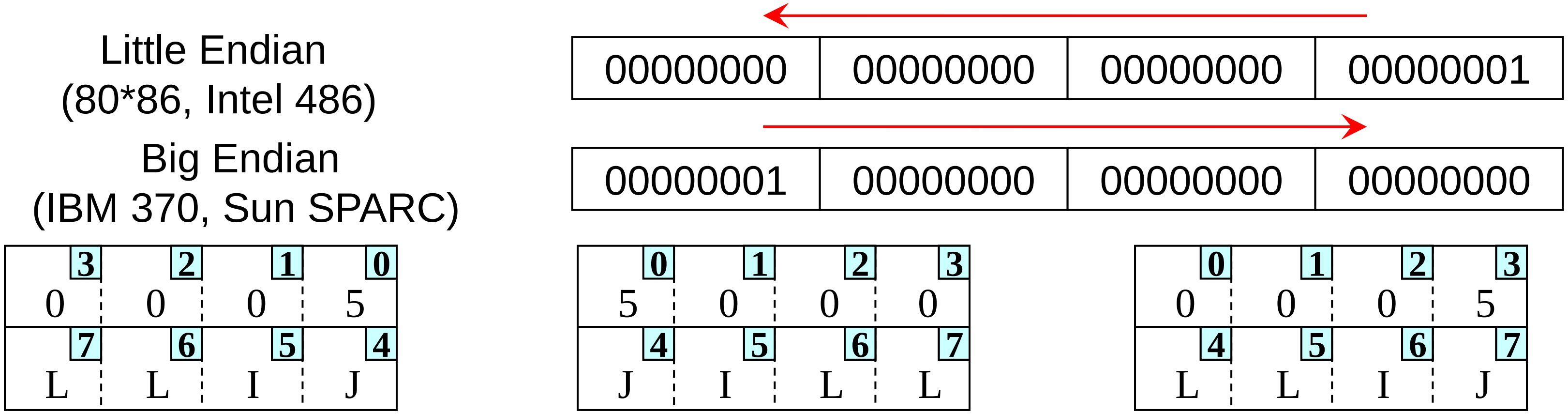
- Parameter passing by value
  - Compose messages that consist of structures for values
  - Problem: Multiple machine types in a distributed system. Each often has its own representation for data! ⇒ Agree on data representation
- Parameter passing by reference
  - Difficult
  - Use copy/restore



Parameter passing by value

Data Representation

- Not a problem when processes are on the same machine
- Different machines have different representations
- Examples:
  - Characters: IBM Mainframes use EBCDIC character code and IBM PCs use ASCII. It is not possible to pass a character from an IBM PC client to an IBM Mainframe server.
  - Integers: Different computers may store bytes of an integer in different order in memory (e.g.,: little endian, big endian), called “host byte order”



Little Endian (80*86, Intel 486)	00000000	00000000	00000000	00000001
Big Endian (IBM 370, Sun SPARC)	00000001	00000000	00000000	00000000

3	2	1	0
0	0	0	5
7	**6**	**5**	**4**
L	L	I	J

Original message on Intel 486

0	1	2	3
5	0	0	0
4	**5**	**6**	**7**
J	I	L	L

Message after receipt on SPARC

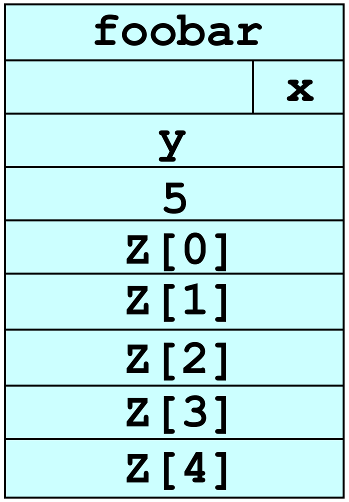
0	1	2	3
0	0	0	5
4	**5**	**6**	**7**
L	L	I	J

Message after being inverted

The Way Out

- Additional information needed about integers and strings
  - Available, since the items in the message correspond to the procedure identifier and the parameters.
- Once a standard has been agreed upon for representing each of the basic data types, given a parameter list and a message, it is possible to deduce which bytes belong to which parameter, and thus to solve the problem
- Given these rules, the requesting process knows that it must use this format, and the receiving process knows that incoming message will have this format
- Having the type information of parameters makes it possible to make necessary conversions

```
foobar (x,y,z)
   char x;
   float y;
   int z[5];
{
…
}
```



foobar

x

y

5

Z[0]

Z[1]

Z[2]

Z[3]

Z[4]

Data Transmission Format

- Option 1: Devise a canonical form for data types and require all senders to convert their internal representation to this form while marshalling.
  - Example: Use ASCII for characters, 0 (false) and 1 (true) for Booleans, etc., with everything stored in little endian
  - Consequence: receiver no longer needs to worry about the byte ordering employed by the sender
  - Sometimes inefficient: Both processes talking big endian; unnecessary conversion at both ends!
- Option 2: Sender uses its own native format and indicates in first byte of message which format is used. Receiver stub examines first byte to see what the client is. Everyone can convert from everyone else’s format.
  - Everyone has to know everyone else’s format
  - Insertion of a new machine with another format is expensive (luckily, does not happen too often)
  - CORBA uses this approach

RPC in Presence of Failures

- Five different classes of failures can occur in RPC systems
  - The client is unable to locate the server
  - The request message from the client to the server is lost
  - The reply message from the server to the client is lost
  - The server crashes after receiving a request
  - The client crashes after sending a request

Client Cannot Locate the Server

- Examples:
  - Server might be down
  - Server evolves (new version of the interface installed and new stubs generated) while the client is compiled with an older version of the client stub
- Possible solutions:
  - Use a special code, such as "-1", as the return value of the procedure to indicate failure. In Unix, add a new error type and assign the corresponding value to the global variable `errno`.
    - → "-1" can be a legal value to be returned, e.g., `sum(7, -8)`
  - Have the error raise an exception (like in ADA) or a signal (like in C).
    - → Not every language has exceptions/signals (e.g., Pascal). Writing an exception/signal handler destroys the transparency

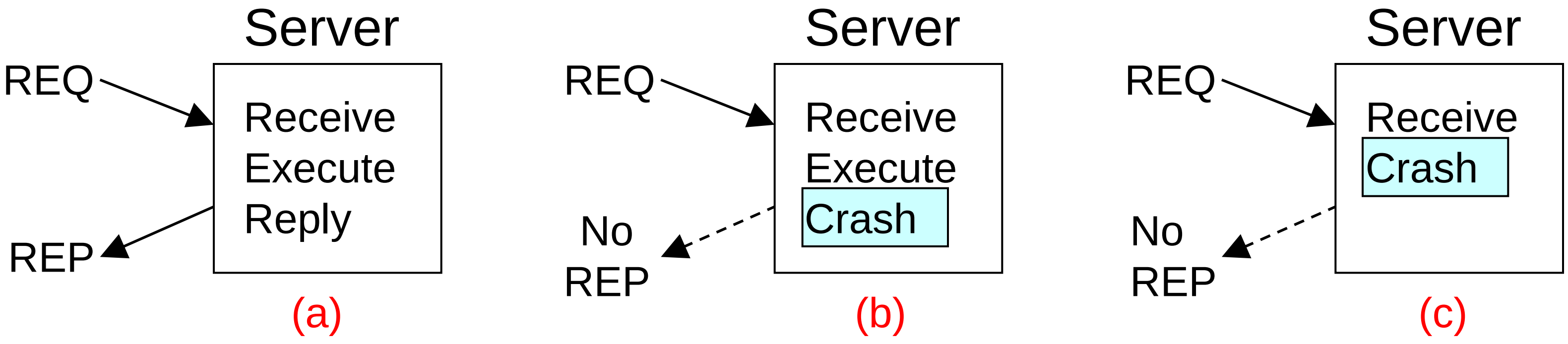
Lost Request Message

- Kernel starts a timer when sending the message:
  - If timer expires before reply or ACK comes back: Kernel retransmits
  - If message truly lost: Server will not differentiate between original and retransmission ⇒ everything will work fine
  - If many requests are lost: Kernel gives up and falsely concludes that the server is down ⇒ we are back to “Cannot locate server”

Lost Reply Message

- Kernel starts a timer when sending the message:
  - If timer expires before reply comes back: Retransmits the request
    - → Problem: Not sure why no reply (reply/request lost or server slow) ?
  - If server is just slow: The procedure will be executed several times
    - → Problem: What if the request is not idempotent, e.g. money transfer
  - Way out: Client's kernel assigns sequence numbers to requests to allow server's kernel to differentiate retransmissions from original

Server crashes



- Problem: Clients’ kernel cannot differentiate between (b) and (c)

  (Note: Crash can occur before Receive, but this is the same as (c).)

- 3 schools of thought exist on what to do here:
  - Wait until the server reboots and try the operation again. Guarantees that RPC has been executed at least one time (at least once semantic)
  - Give up immediately and report back failure. Guarantees that RPC has been carried out at most one time (at most once semantics)
  - Client gets no help. Guarantees nothing (RPC may have been carried out anywhere from 0 to a large number). Easy to implement.

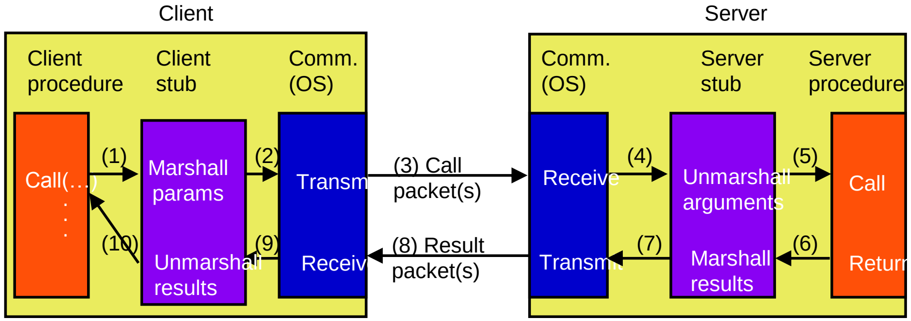
Client Crashes

- Client sends a request and crashes before the server replies: A computation is active and no parent is waiting for result (orphan)
  - Orphans waste CPU cycles and can lock files or tie up valuable resources
  - Orphans can cause confusion (client reboots and does RPC again, but the reply from the orphan comes back immediately afterwards)
- Possible solutions
  - Extermination: Before a client stub sends an RPC, it makes a log entry (in safe storage) telling what it is about to do. After a reboot, the log is checked and the orphan explicitly killed off.
    - → Expense of writing a disk record for every RPC; orphans may do RPCs, thus creating grandorphans impossible to locate; impossibility to kill orphans if the network is partitioned due to failure.

Client Crashes (2)

- Possible solutions (cont'd)
  - Reincarnation: Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring the start of a new epoch. When broadcast comes, all remote computations are killed. Solve problem without the need to write disk records
    - → If network is partitioned, some orphans may survive. But, when they report back, they are easily detected given their obsolete epoch number
  - Gentle reincarnation: A variant of previous one, but less Draconian
    - → When an epoch broadcast comes in, each machine that has remote computations tries to locate their owner. A computation is killed only if the owner cannot be found
  - Expiration: Each RPC is given a standard amount of time, *T*, to do the job. If it cannot finish, it must explicitly ask for another quantum.
    - → Choosing a reasonable value of *T* in the face of RPCs with wildly differing requirements is difficult

RPC Recap



- Logical extension of the procedure call interface
  - Easy for the programmer to understand and use
- Requires an Interface Definition Language (IDL) to specify data types and procedure interfaces
  - Provides language independence
- IDL compiler generates client and server stubs
  - Stubs handle the marshalling and unmarshalling of arguments and builds the message
    - → Messages must be represented in a machine independent data representation
    - → Must flatten or serialize complex types
- Calling and called procedures reside in different address space
  - Cannot send pointers or system specific data (locks, file descriptors, pipes, sockets, etc.)
  - Parameter passing can therefore be very expensive
  - Breaks transparency!

RPC Recap

- New failure models
  - Cannot locate server (throw exception)
  - Lost request message (resend request)
  - Lost reply message
    - → Ack reply and resend reply if ack not received
      - – What if network failure is sufficiently long that your TCP connection times out?
    - → Retransmit request without incrementing the request sequence number
    - → Requires that the server retain old replies for some set time period
  - Server crashes before sending reply
    - → RPC resends request (at least once semantics)
    - → Give up and report failure (at most once semantics)
      - – Or server saves all replies on persistent storage and re-send previous replies
    - → Client gets no help, semantics determined by the client
  - Client crashes

RPC Example: Apache Thrift

- You will be using this for your project
- Developed at Facebook and open-sourced in 2007
- Specifies its own IDL

```
struct UserProfile {
    1: i32 uid,
    2: string name,
    3: string blurb
}
```

```
service UserStorage {
    void store(1: UserProfile user),
    UserProfile retrieve(1: i32 uid)
}
```

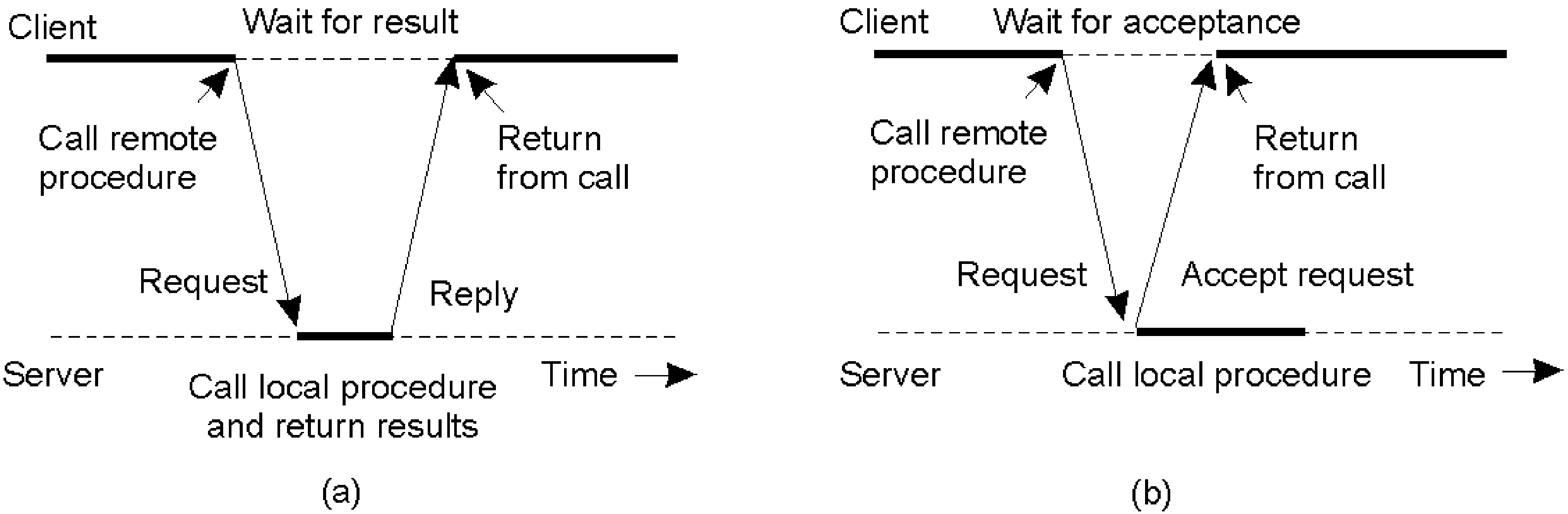
- IDL compiler generates client/server stubs
  - Server has to implement the handler for the service

```
class UserStorageHandler : virtual public UserStorageIf {
    public: UserStorageHandler() {
        // Your initialization goes here
    }
    void store(const UserProfile& user) {
        // Your implementation goes here
    }
    void retrieve(UserProfile& _return, const int32_t uid) {
        // Your implementation goes here
    }
};
```

RPC Example: Apache Thrift

- Supports
  - C++, C#, Obj-C, Erlang, Haskell, Java, OCaml, Perl, PHP, Python, Ruby, Smalltalk
- Different data encoding types
  - Binary, JSON
- IDL provides relatively simple datatypes
  - Primitives
  - Simple structures (structs, lists, maps)
- At most once semantics
- Does not provide any solution for orphans

Asynchronous RPC

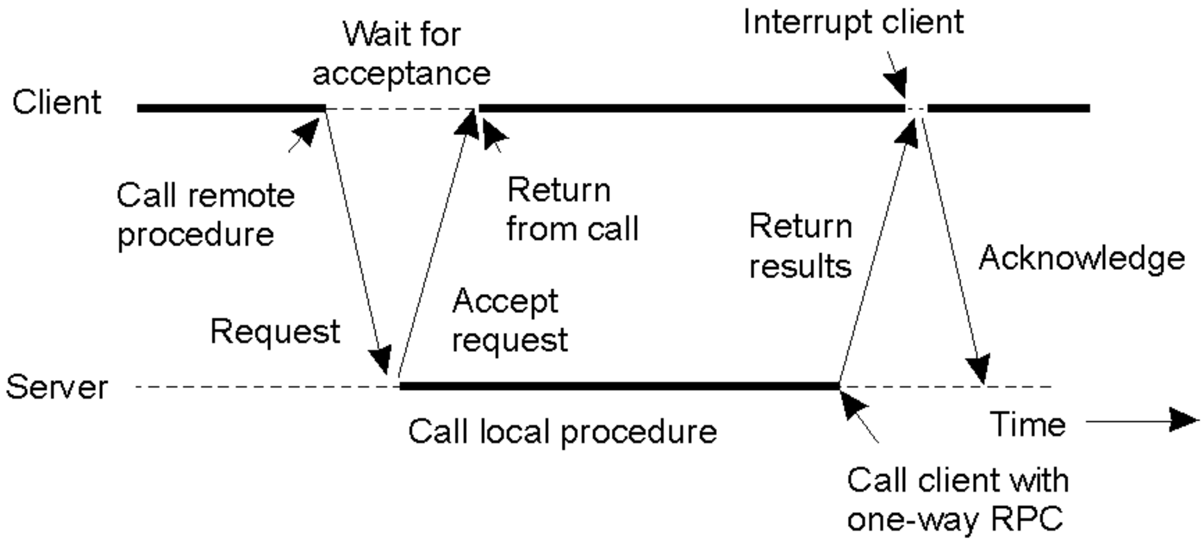


Traditional (synchronous) RPC interaction

Asynchronous RPC interaction

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

Asynchronous RPC (2)



- A client and server interacting through two asynchronous RPCs
  - Known as deferred synchronous RPC
- Some RPCs do not require that the client waits for the acceptance
  - Known as one-way RPC

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

How does the client locate the server?

- Hardwire the server address into the client
  - Fast but inflexible!
- A better method is dynamic binding:
  - When the server begins executing, the call to initialize outside the main loop exports the server interface
  - This means that the server sends a message to a program called a binder, to make its existence known. This process is referred to as registering
  - To register, the server gives the binder its name, its version number, a unique identifier (32-bits), and a handle used to locate it
  - The handle is system dependent (e.g., Ethernet address, IP address, an X.500 address, …)

	Call	Input	Output
Binder interface	Register	Name, version, handle, unique id	
	Deregister	Name, version, unique id	

Dynamic Binding

- When the client calls one of the remote procedures for the first time, say, read:
  - The client stub sees that it is not yet bound to a server, so it sends a message to the binder asking to import version x of server interface
  - The binder checks to see if one or more servers have already exported an interface with this name and version number.
    - → If no currently running server is willing to support this interface, the read call fails
    - → If a suitable server exists, the binder gives its handle and unique identifier to the client stub
  - Client stub uses the handle as the address to send the request message to. The message contains the parameters and a unique identifier, which the server's kernel uses to direct incoming message to the correct server

	Call	Input	Output
Binder interface	Look up	Name, version	Handle, unique id

Dynamic Binding (2)

- Advantages
  - Flexibility
  - Can support multiple servers that support the same interface, e.g.:
    - → Binder can spread the clients randomly over the servers to even load
    - → Binder can poll the servers periodically, automatically deregistering the servers that fail to respond, to achieve a degree of fault tolerance
    - → Binder can assist in authentication: For example, a server specifies a list of users that can use it; the binder will refuse to tell users not on the list about the server
  - The binder can verify that both client and server are using the same version of the interface
- Disadvantages
  - The extra overhead of exporting/importing interfaces costs time
  - The binder may become a bottleneck in a large distributed system

Distributed Object Systems

- Most large enterprise applications are written in an OO language and utilize an OO paradigm to manage complexity.
- Naturally, in place of RPCs, we have distributed objects and Remote Method Invocation.
- Provides transparency for OO applications

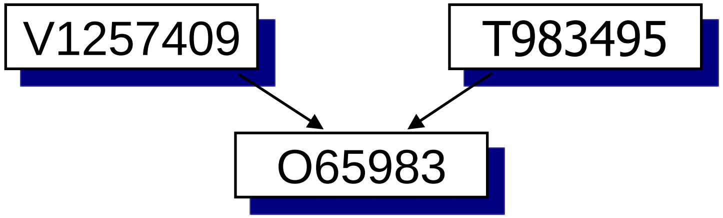
Distributed Object Systems

- Also introduces a lot of new complications
  - In RPC, a server exposes a small number of unique procedures. With distributed objects, each object instantiation is unique.
    - → A distributed object system must provide a globally unique identifier for each object
  - Remote object references can be passed as parameters
    - → RMI to an object can be initiated from multiple clients
    - → How do we handle concurrent access to a remote object?
  - Objects can migrate and are sometimes replicated
    - → How do we ensure object references point to the correct object no matter where it is located?
    - → How do we ensure consistency between replicated objects?
  - Many OO languages provide garbage collection
    - → How do we provide distributed garbage collection?

Object Identity (OID) & Referencing

- OID is an invariant property of an object which distinguishes it logically and physically from all other objects.
  - OIDs are unique.
  - OIDs are independent of state.
- Time invariance
  - OID:O1
    → VIN: “V1257409”
    → Make: “Volvo”
    → …
    → Owner: O65983

    OID: O1
    VIN: “V1257409”
    Make: “Volvo”
    …
    Owner: O97603
- Referential sharing
  - Multiple objects share common sub-objects



V1257409

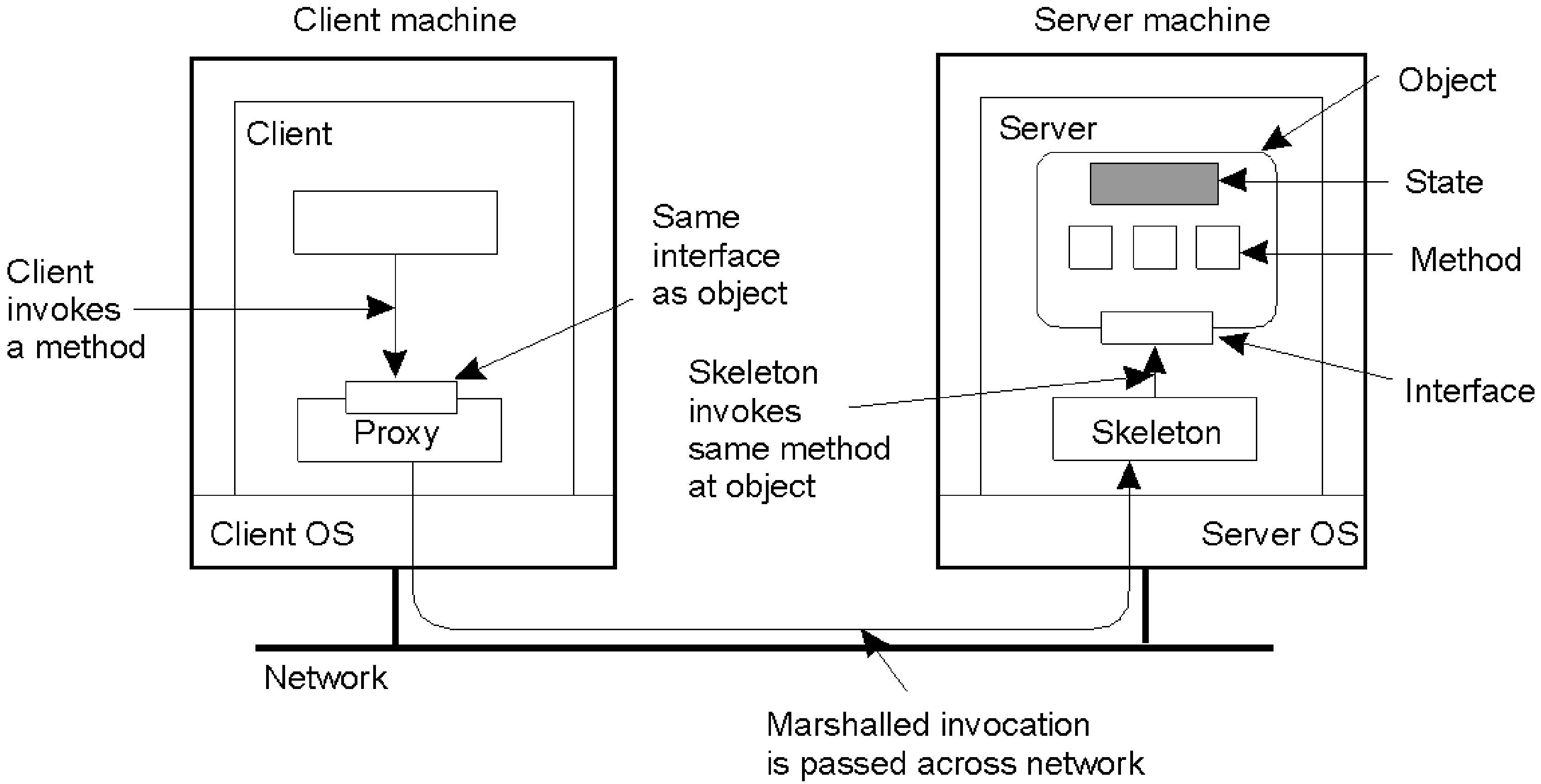
T983495

O65983

Object References

- Should encode:
  - Which object is being referenced
  - The location of the object
  - RMI specific details
    - Which protocol is used (TCP/UDP/HTTP)
    - How is the parameters being marshalled?
      - Binary, XML, JSON, etc.
- The object reference can even include an implementation handle
  - Specifies the location of the object proxy implementation, which serves the same role as the client stub in RPCs.
  - Enable clients to dynamically add support to protocols and marshalling mechanisms.
  - Can even enable a client to dynamically make use of objects that it wasn't aware of at compile time.

RMI Implementation



From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

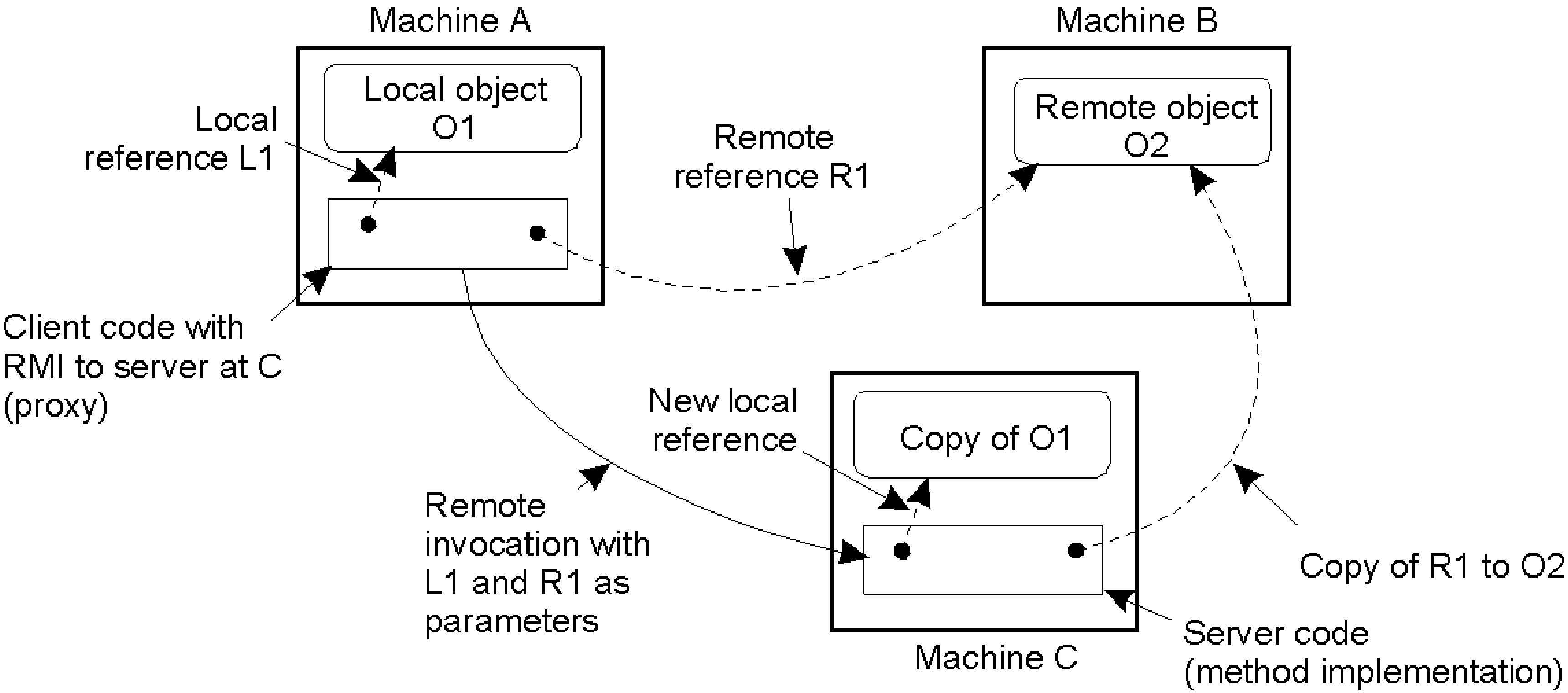
Object Identity and Distribution

- Logical versus physical OIDs
  - LOID references an object via indirection.
    - → More flexible.
  - POID indicates the physical address of an object.
    - → Better performance.
  - If LOIDs are used, they need to be mapped to POIDs at some point.
- Must address the impact of migration, replication, and, for data objects, fragmentation on OIDs
- Must generate OIDs in a distributed manner
  - For LOIDs: Timestamps with server id's
- Must be able to reuse OIDs

Location Server

- Rather than encode the address of the server hosting the object, encode a system-wide identifier for the server together with the address of a separate location server
- Object hosting servers register their location with their identifier at the location server
- Location server provides an indirection service
- Transplanting the server to a different machine is transparent to the client
  - Only requires changing the identifier to address mapping
- Problem:
  - Location server becomes the bottleneck and creates a single-point of failure
  - Can piggyback on top of a distributed naming system
    - DNS
    - Distributed hash table

Parameter Passing



The situation when passing an object by reference or by value.

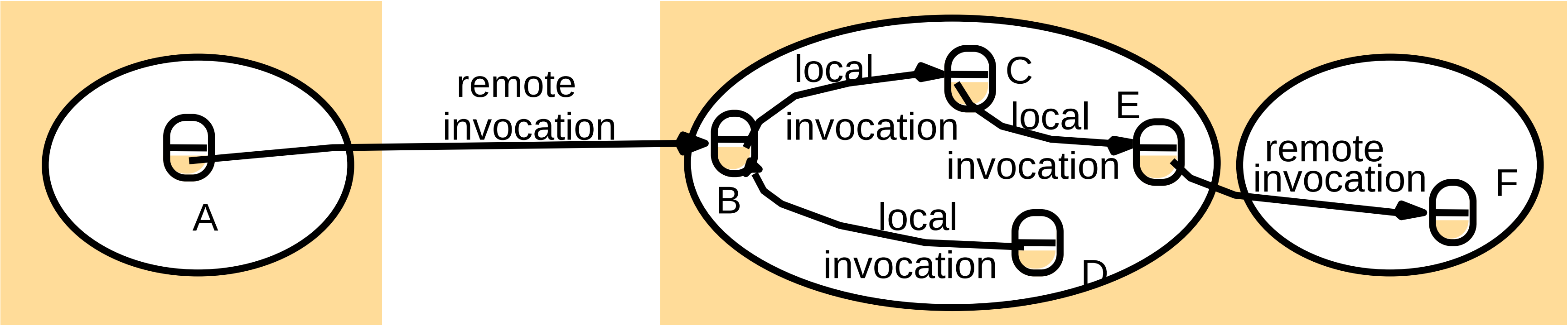
From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

Distributed Garbage Collection

- A Global Mark and Sweep is generally not an option
  - Where is the root object?
  - Pausing a large-scale distributed system to perform garbage collection is generally not acceptable
- Java RMI uses reference counting
  - When an live reference enters a JVM, the JVM sends a “referenced” message to the server and the objects reference count is incremented.
  - When the last reference to a remote object in a JVM is discarded, an “unreferenced” message is sent and the reference count is decremented.
  - If the reference count is 0 and the JVM hosting the object no longer has a reference to it, the object is garbage collected.
  - Requires careful handling of ordering of referenced and unreferenced messages across JVMs.
    - If JVM_1 passes off a remote reference to OBJ_0 to JVM_2 and then sends a “unreferenced” message to JVM_0 (hosting the object) before JVM_2’s “referenced” message arrives at JVM_0, it can cause JVM_0 to prematurely garbage collect OBJ_0.
  - Cannot handle circular references
  - Network partitions can also cause premature garbage collection.

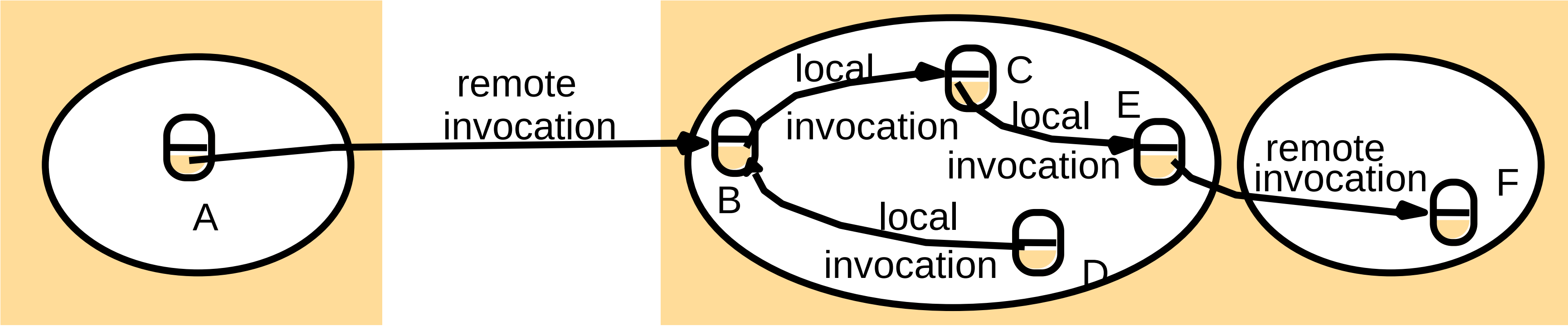
Object Distribution Model

- We will use a simple distribution model
  - Adoption of client- server model
    - Servers maintain objects and allow remote objects to access the methods of those objects that are specified for remote use in the interface definition
    - Clients invoke the methods offered by servers using remote method invocation (RMI)
  - RMI implemented using a request/ reply protocol
    - Request carries remote method reference and remote object’s in parameters to server
    - Reply returns out parameters (results of RMI) to client



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

Object Distribution Issues



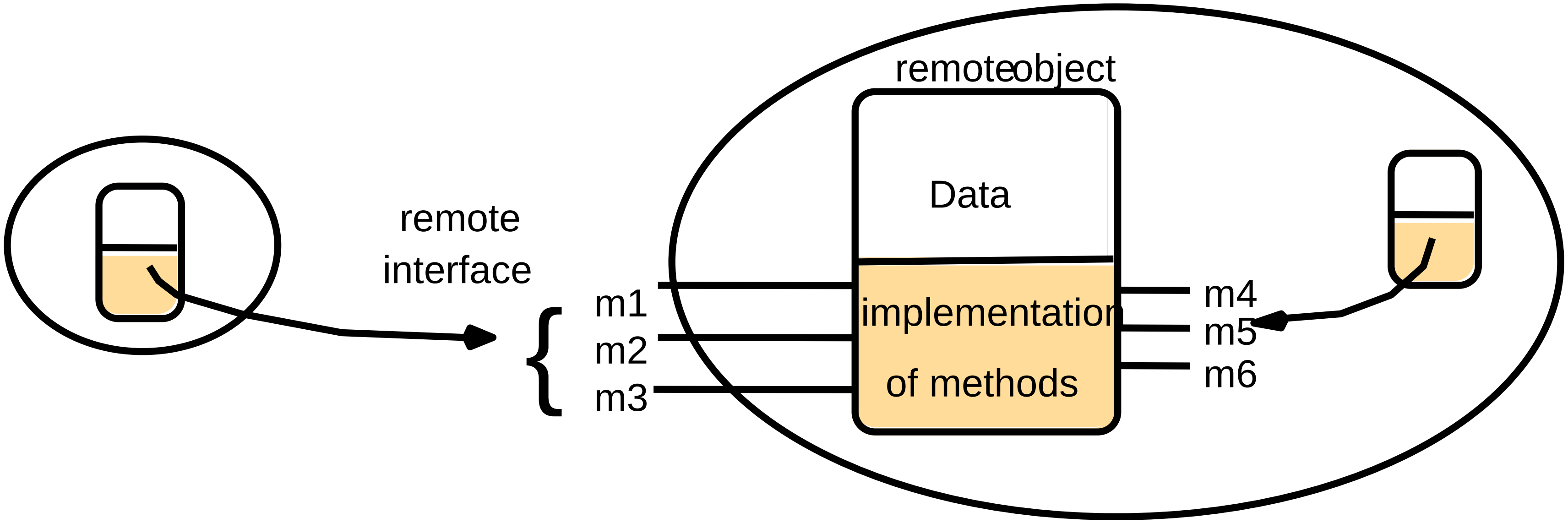
- Remote Object
  - object capable of receiving RMIs (in example at least B and F)
- Remote Object Reference
  - needed by an object performing a RMI
  - refers to server of RMI
  - typical format

32 bits	32 bits	32 bits	32 bits	
Internet address	Port number	Time	Object number	Interface of remote object

  - can be passed as parameter or result

Remote Interface

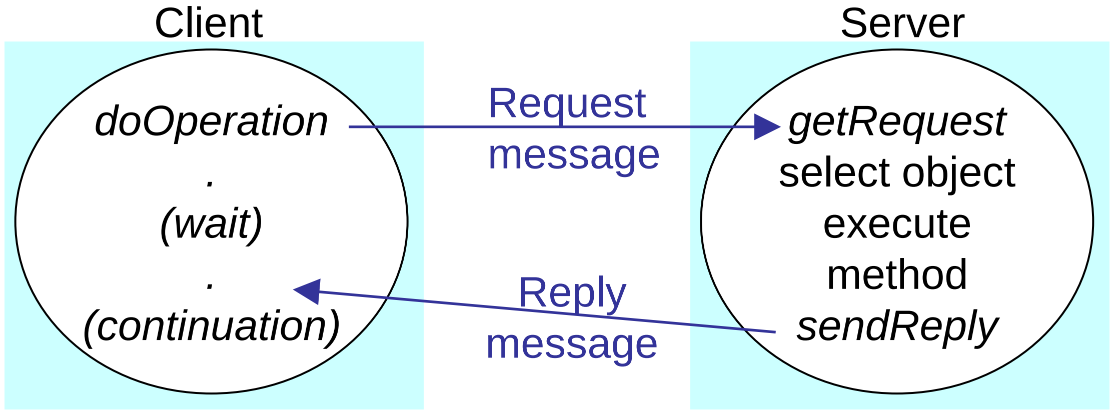
- Specifies which methods can be invoked remotely



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

RMI Invocation Semantics

- RMI usually implemented via request/reply protocol
  - Recall discussion at the beginning of this module



public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply. The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message to the client at its Internet address and port.

RMI Invocation Semantics (2)

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

- Maybe
  - Invoker knows nothing about whether operation is executed
  - No fault tolerance mechanism used

RMI Invocation Semantics (3)

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

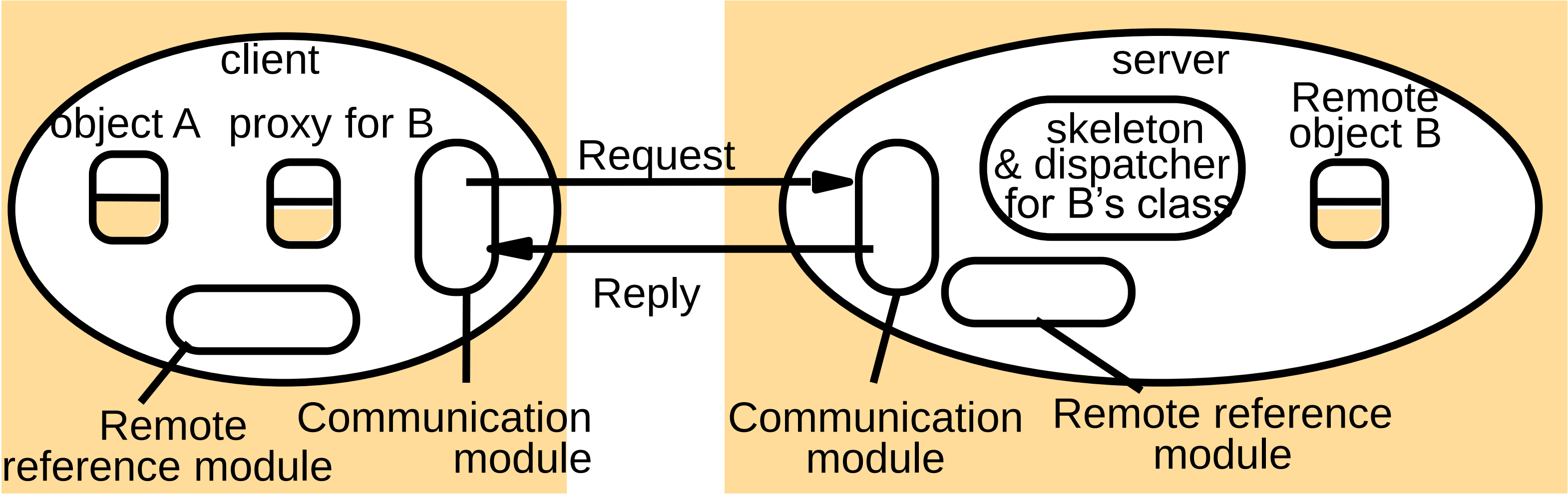
- At-least-once
  - Invoker receives result, or exception, if no result was received
  - If result received, invoker knows that operation was at least executed once, but knows nothing if exception occurs
  - Achieved through retransmissions of requests and re-executions
  - Can suffer from the following types of failures
    - Crash failure of server containing remote object
    - Arbitrary failures caused through re- execution of non-idempotent operation on server

RMI invocation semantics (4)

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

- At-most-once
  - Invoker receives
    - → result, in which case s/he knows that operation was executed exactly once, or
    - → exception, informing that no result was received, in which case the operation was either performed once, or not at all
  - Requires usage of all fault tolerance measures
  - Examples
    - → CORBA and Java RMI have at most once semantic

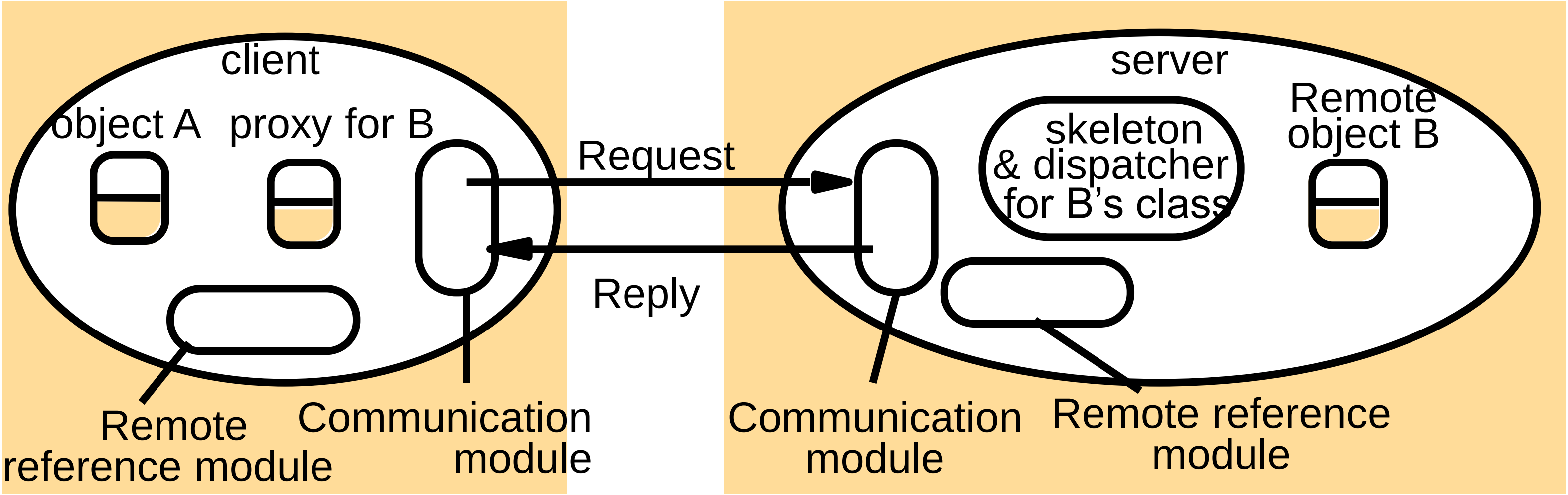
RMI Implementation



- Communication Module
  - request/ reply protocol
  - uses message type, requestID and remote object reference
  - in server
    - select dispatcher for the class of the object to be invoked
    - obtains local reference from remote reference module by providing remote object identifier delivered with request message
    - passes on the local reference to dispatcher

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

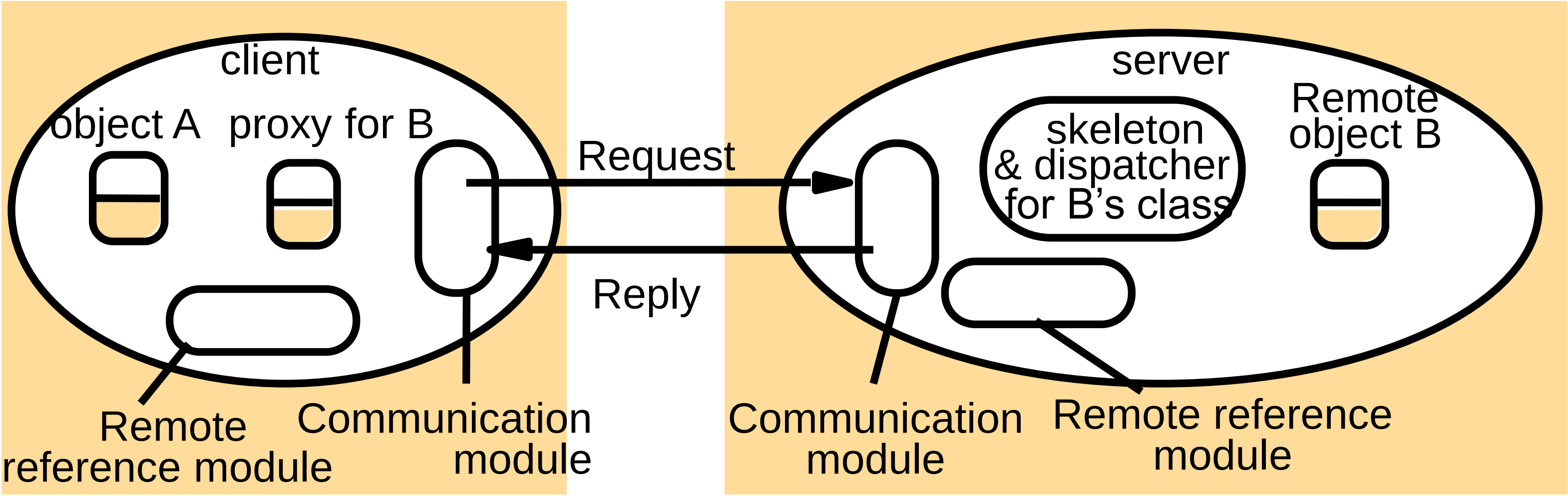
Implementation of RMI



- Remote Reference Module
  - Translation between local and remote object references using remote object table
    - contains entries for all remote objects maintained by the server
    - and for all proxies (explained later)
  - Used when marshalling and unmarshalling remote object references

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

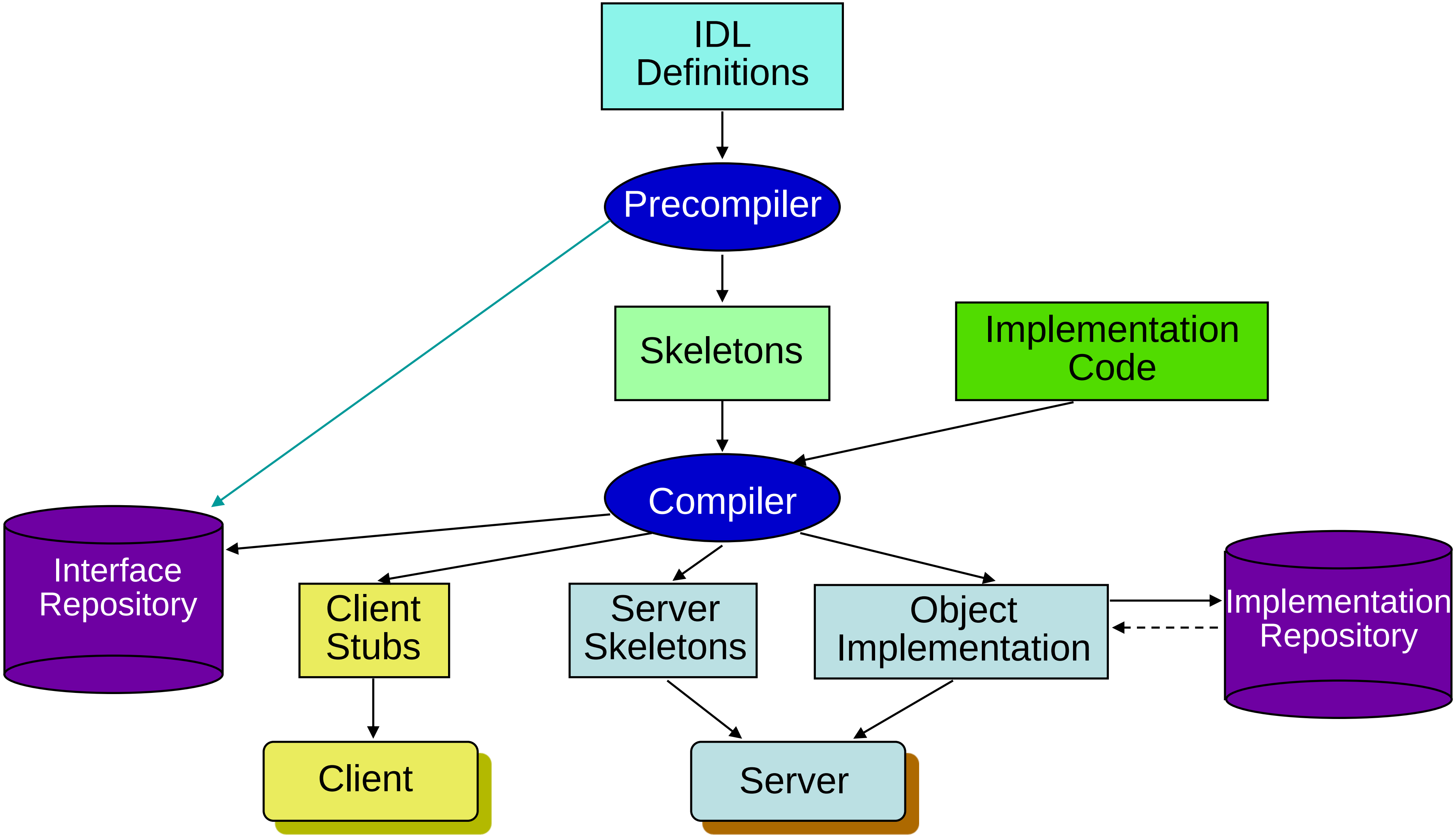
Implementation of RMI



- The RMI sublayer of the middleware architecture consists of:
  - proxies (in client): local placeholder for remote objects
  - dispatcher (in server): receives request and uses methodID to select appropriate method in skeleton
  - skeleton (in server): implements methods in the remote interface
    - unmarshaling arguments
    - invokes corresponding method in remote object
    - waits for completion of invocation
    - marshals results, possibly exceptions, and returns them in a reply to invoking method of proxy

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

CORBA Stub & Skeleton Generation



Some more RMI components

- Factory methods
  - Remote object interfaces cannot include constructors
  - Factory methods replace constructors to create remote objects
- Binders
  - Service maintaining mappings from textual object names to remote object references
- Threads in server
  - Every remote method invocation is usually given an own thread to execute
    - → Advantages, if RMI invokes other local or remote methods that cause wait times

Message-Based Communication

- Lower-level interface to provide more flexibility
- Two (abstract) primitives are used to implement these
  - send
  - receive
- Issues:
  - Are primitives blocking or nonblocking (synchronous or asynchronous)?
  - Are primitives reliable or unreliable (persistent or transient)?

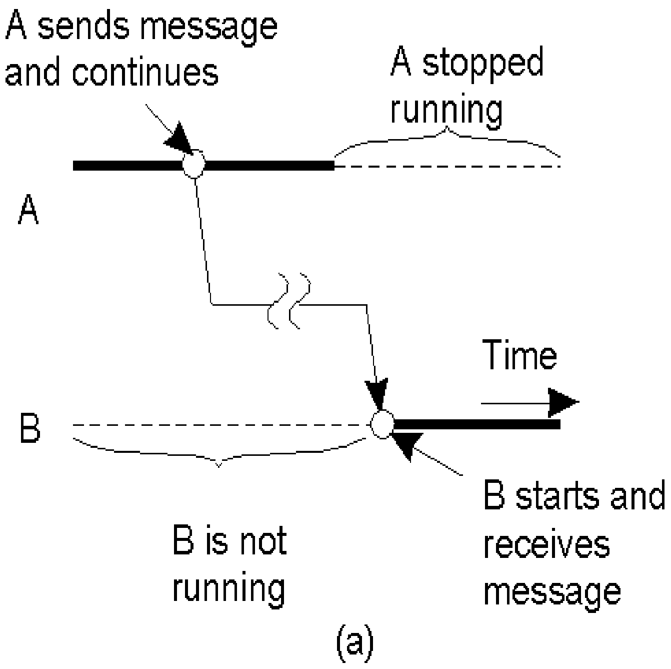
Synchronous/Asynchronous Messaging

- Synchronous
  - The sender is blocked until its message is stored in the local buffer at the receiving host or delivered to the receiver.
- Asynchronous
  - The sender continues immediately after executing a send
  - The message is stored in the local buffer at the sending host or at the first communication server.

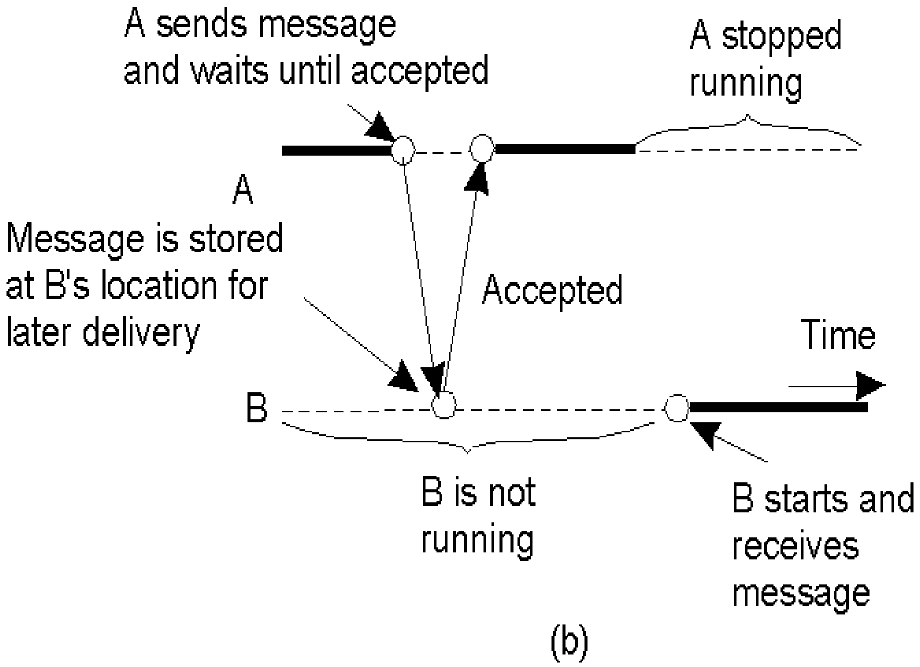
Transient/Persistent Messaging

- Transient
  - The sender puts the message on the net and if it cannot be delivered to the sender or to the next communication host, it is lost.
  - There can be different types depending on whether it is asynchronous or synchronous
- Persistent
  - The message is stored in the communication system as long as it takes to deliver the message to the receiver

Persistent Messaging Alternatives



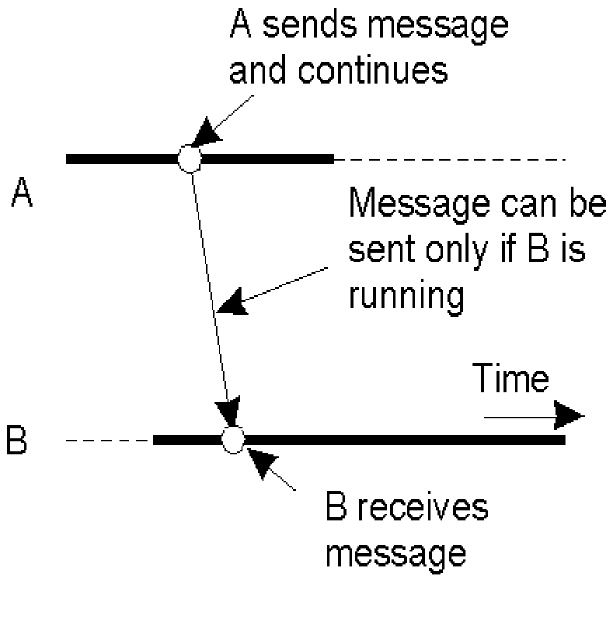
Persistent asynchronous communication


A sends message and waits until accepted
A stopped running
A
Message is stored at B's location for later delivery
Accepted
Time
B
B is not running
B starts and receives message
(b)

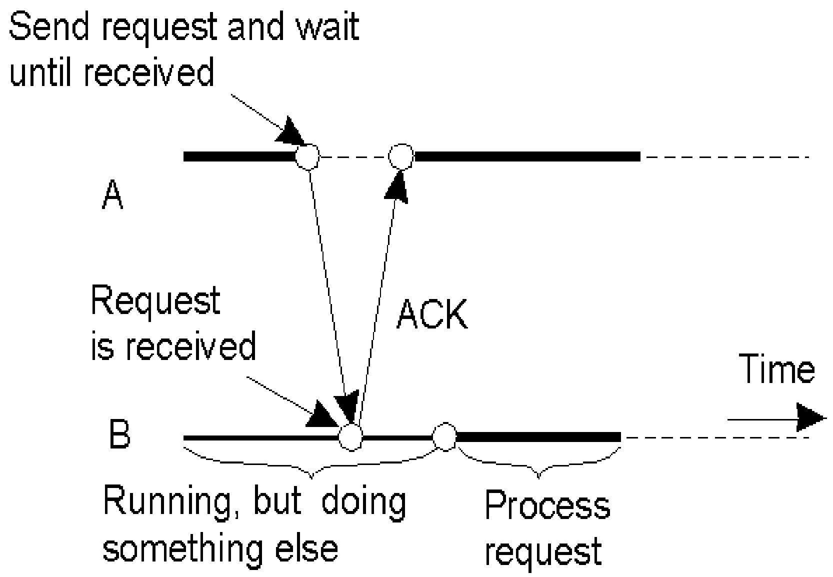
Persistent synchronous communication

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

Transient Messaging Alternatives


A sends message and continues
A
Message can be sent only if B is running
Time
B
B receives message

(c)


Send request and wait until received
A
Request is received
ACK
Time
B
Running, but doing something else
Process request

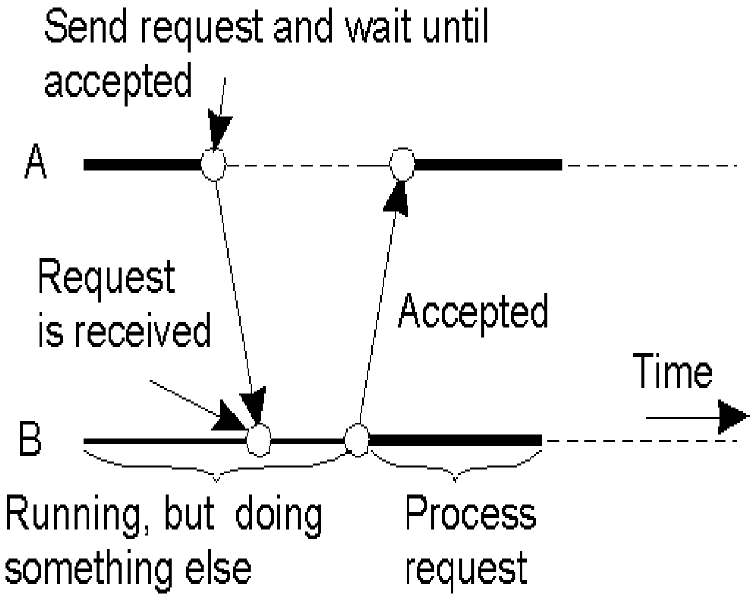
(d)

Transient asynchronous communication

Receipt-based transient synchronous communication

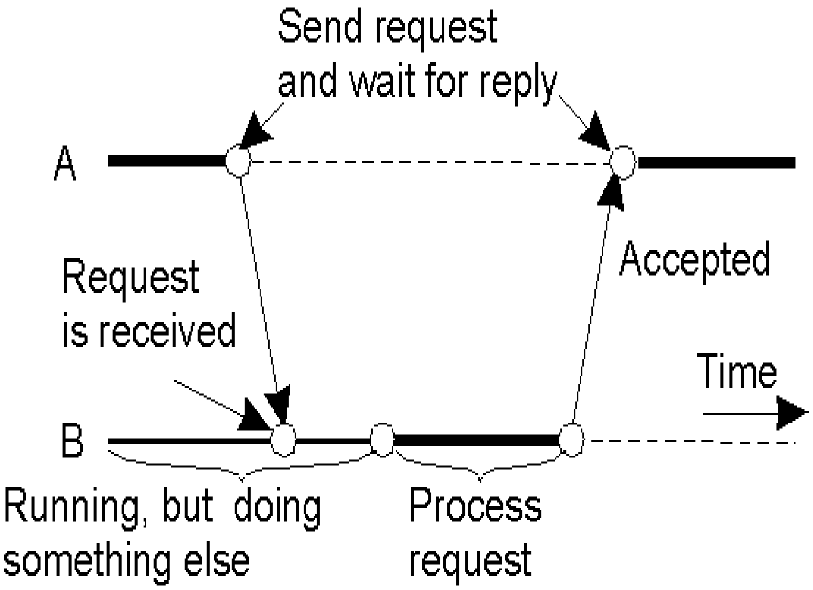
From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms

Transient Messaging Alternatives (2)


Send request and wait until accepted
A
Request is received
Accepted
Time
B
Running, but doing something else
Process request

(e)

Delivery-based transient synchronous communication at message delivery


Send request and wait for reply
A
Request is received
Accepted
Time
B
Running, but doing something else
Process request

(f)

Response-based transient synchronous communication

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

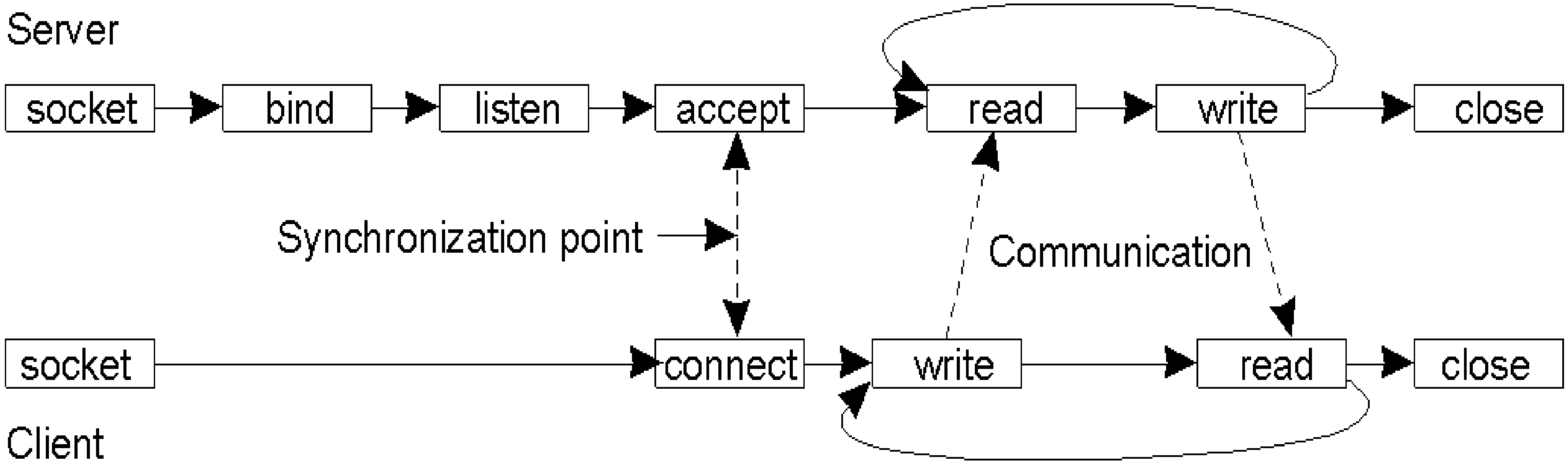
Transient Messaging

- Example: Socket primitives for TCP/IP.

Primitive	**Meaning**
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms

Connection-Oriented Communication Using Sockets



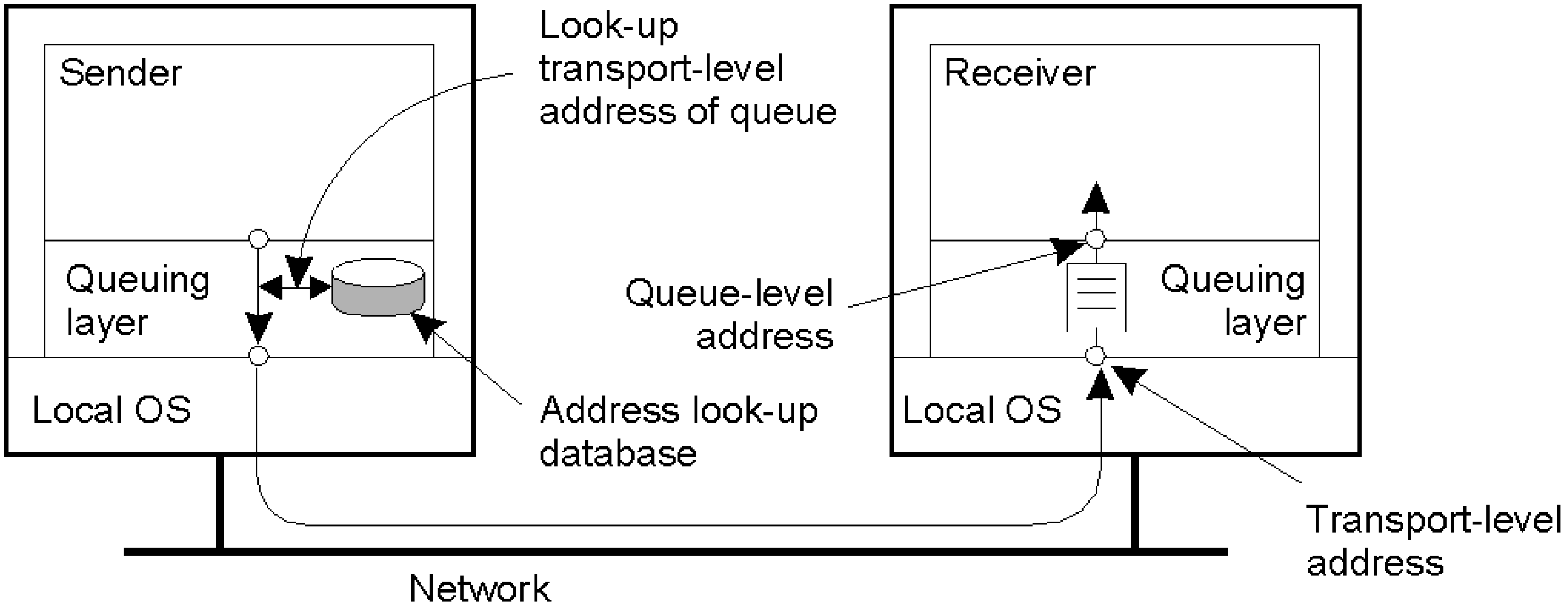
From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms

Persistent Messaging

- Usually called message-queuing system, since it involves queues at both ends
  - Sender application has a queue
  - Receiver application has a queue
  - Receiver does not have to be active when sender puts a message into sender queue
  - Sender does not have to be active when receiver picks up a message from its queue
- Basic interface:

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.

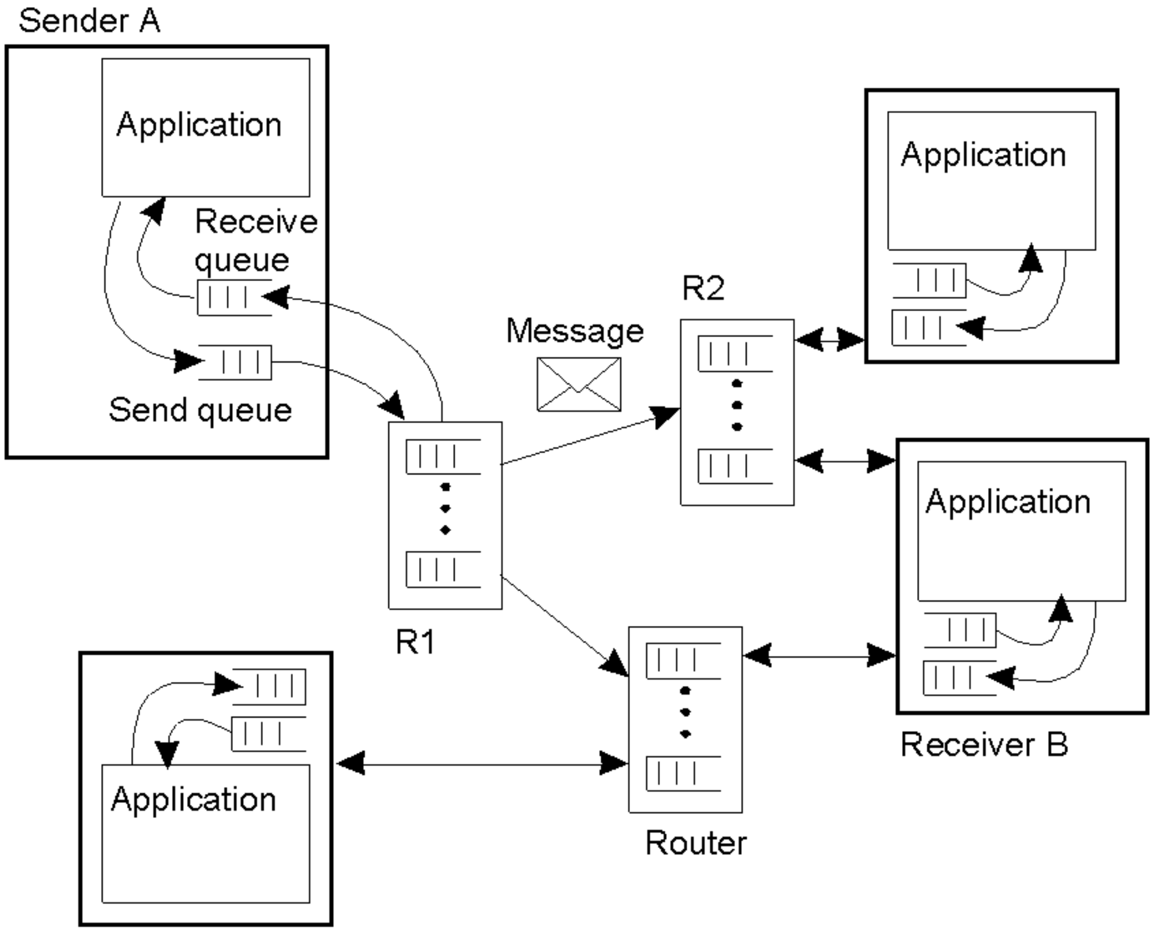
General Architecture of a Message-Queuing System (1)



From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

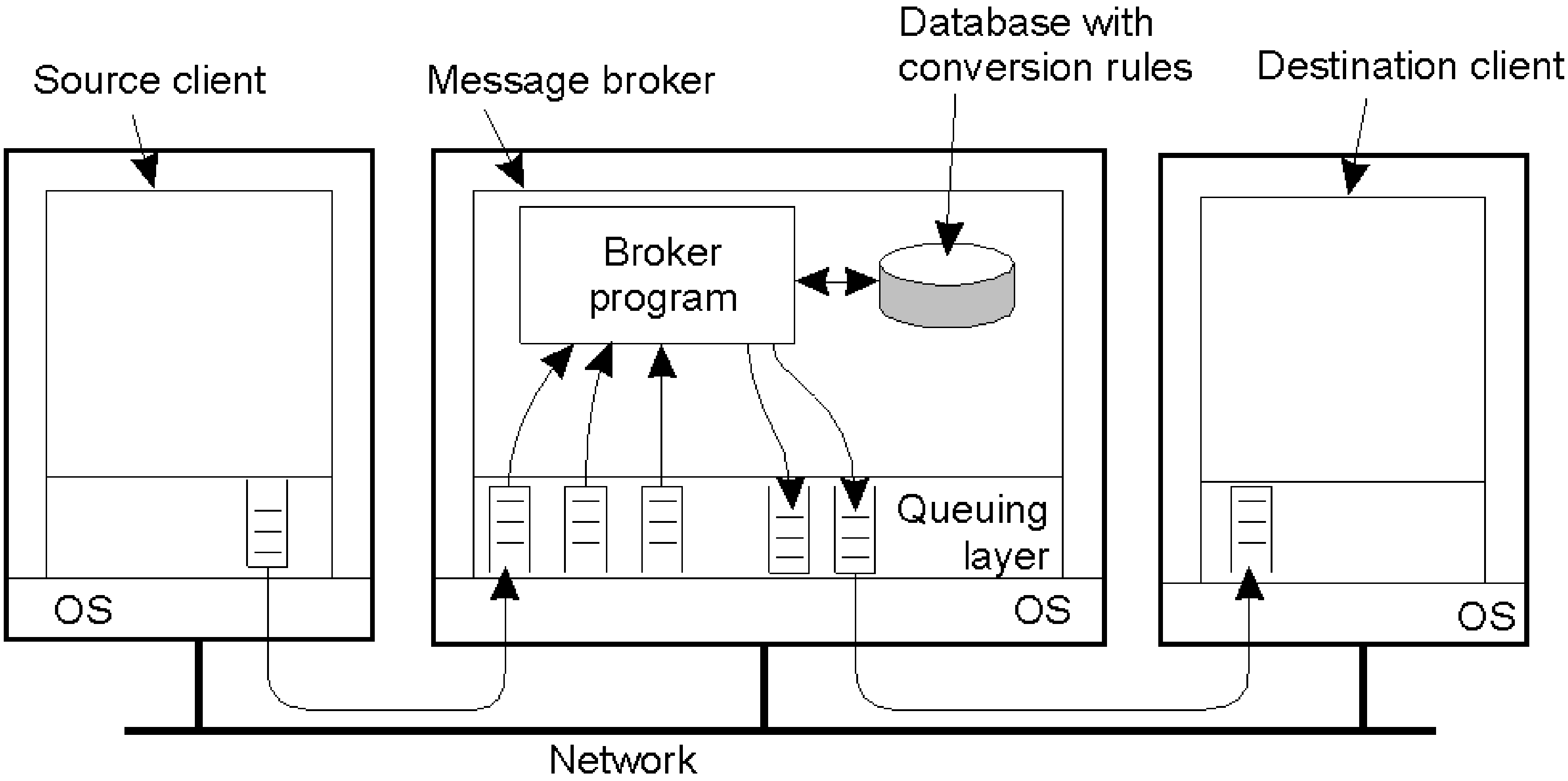
General Architecture of a Message-Queuing System (2)

- The geı



From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

Message Brokers



From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002