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

```plaintext
//In file Person.idl
Struct Person {
  string name;
  string place;
  long year;
};

Interface 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
  \[
  \text{count} = \text{read}(\text{fd}, \text{buf}, \text{nbytes})
  \]
  where \text{fd} (file descriptor) and \text{nbytes} (no. bytes to read) are integers, and \text{buf} (buffer into which data are read) is an array of characters

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.
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

```
Kernel

n = sum(4, 7);
```

```
Kernel

sum(i, j)
int i, j;
{
    return(i + j);
}
```

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)

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```
Big Endian
(IBM 370, Sun SPARC)

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

- Original message on Intel 486
- Message after receipt on SPARC
- 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];
{
...
}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>z[0]</td>
<td></td>
</tr>
<tr>
<td>z[1]</td>
<td></td>
</tr>
<tr>
<td>z[2]</td>
<td></td>
</tr>
<tr>
<td>z[3]</td>
<td></td>
</tr>
<tr>
<td>z[4]</td>
<td></td>
</tr>
</tbody>
</table>

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

Asynchronous RPC

- Traditional (synchronous) RPC interaction
- Asynchronous RPC interaction

Asynchronous RPC (2)

Where do stub procedures come from?

- In many systems, stub procedures are generated automatically by a special stub compiler.
- Given a specification of the server procedure and the encoding rules, the message format is uniquely determined.
- It is, thus, possible to have a compiler read the server specification and generate a client stub that packs its parameters into the officially approved message format.
- Similarly, the compiler can also produce a server stub that unpacks the parameters and calls the server.

Having both stub procedures generated from a single formal spec. of the server not only makes life easier for the programmers, but reduces the chance of error and makes the system transparent with respect to differences in internal representation of data.
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, …)

<table>
<thead>
<tr>
<th>Binder interface</th>
<th>Call</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Name, version, handle, unique id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deregister</td>
<td>Name, version, unique id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Call</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look up</td>
<td>Name, version</td>
<td>Handle, unique id</td>
</tr>
</tbody>
</table>

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 Management

- **Bad news:**
  - Additional complications
    - Distribution problems
    - Object-orientation problems
      - Object ids
      - Object state
  - Additional complications
  - Distribution problems
  - Object-orientation problems

- **Good news:**
  - A good model for distributed computing
    - Computation is distributed among objects; now assume objects are located on different computers
  - Object-orientation features
    - Encapsulation
    - Abstraction
    - Extensible type system
    - Code reuse (inheritance)

**Object**

- An abstraction defined by:
  - A unique identifier (OID)
  - Valued attributes (instance variables) which give the state of the object
  - An interface in terms of methods (operations) which manipulate the object
  - Some models combine attributes and methods into uniform behaviors
  - Objects can be data abstractions or program (service) abstractions

- **Example:** Car object
  - **State**
    - VIN: “V1257409”
    - Make: “Volvo”
    - Model: “GLT”
    - Year: 1998
    - Owner: 065983
  - **Methods**
    - Age: Thisyear- Year
Types

- Each object has a type. Generally, each object has only one type
  - Example: Car is a type; “V1257409” is an object (instance) of type Car
- Types enable classification: is-a relationship

```
  is-a
  ---->
     Vehicle

     Car ... Truck

     Volvo ... BMW
```

- Generally, the is-a relationship is referred to as subtyping.
  - Car is a subtype of Vehicle; Vehicle is a supertype of Car
  - Subtyping allows substitutability.

Classes

- A class is a grouping of all object instances of a given type. A class is the extent of a given type.
- A class of a subtype is a subset of a class of supertype
  - The set of all Volvo’s is a subtype of the set of all Cars.
Types/classes and Distribution

- Deep extent of a class is a union of subclass extents. Subclasses may reside at different sites.
- Is the type replicated at every site?
  - Class is fragmented to allow storage of instances of a type at different sites.
- At what level is replication and migration of data performed?
  - At the type level
  - At the class level

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
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

Object Distribution

- Location of objects with respect to their types
  - Objects are co-located with their types (different types of objects are located at different sites)
  - Objects of one type can be located at different sites (horizontal fragmentation)
- Replication
  - What is the unit of replication (object, class, collection)
  - Are types duplicated at various sites?
- Object processing
  - Local function - local argument
  - Local function - remote argument
    - Local surrogates (proxies)
  - Remote function - local argument
  - Remote function - remote argument
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

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
    - | Internet address | Port number | Time | Object number | Interface of remote object |
      |------------------|-------------|------|--------------|--------------------------|
      | 32 bits          | 32 bits     | 32 bits | 32 bits       |
  - can be passed as parameter or result
Remote Interface

- Specifies which methods can be invoked remotely

RMI Invocation Semantics

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

```java
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)

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request</td>
<td></td>
</tr>
<tr>
<td>message</td>
<td></td>
</tr>
<tr>
<td>Duplicate filtering</td>
<td></td>
</tr>
<tr>
<td>Re-execute procedure</td>
<td></td>
</tr>
<tr>
<td>or retransmit reply</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

RMI Invocation Semantics (3)

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request</td>
<td></td>
</tr>
<tr>
<td>message</td>
<td></td>
</tr>
<tr>
<td>Duplicate filtering</td>
<td></td>
</tr>
<tr>
<td>Re-execute procedure</td>
<td></td>
</tr>
<tr>
<td>or retransmit reply</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

- **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-exectutions
  - 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)

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request message</td>
<td>Duplicate filtering</td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- **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

RMI Implementation (2)

- 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

© Addison-Wesley Publishers 2000

Implementation of RMI (3)

- 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

© Addison-Wesley Publishers 2000
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

---

**CORBA Stub & Skeleton Generation**

- IDL Definitions → Precompiler
  - Skeletons
  - Implementation Code

- Precompiler
  - Compiler
    - Interface Repository
      - Client Stubs
      - Server Skeletons
      - Object Implementation
        - Implementation Repository
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

---

Binding a Client to an Object

(a) An example with implicit binding using only global references

```c
Distr_object* obj_ref; //Declare a systemwide object reference
obj_ref = ...; // Initialize the reference to a distributed object
obj_ref->do_something(); // Implicitly bind and invoke a method
```

(b) An example with explicit binding using global and local references

```c
Distr_object obj_ref; //Declare a systemwide object reference
Local_object* obj_ptr; //Declare a pointer to local objects
obj_ref = ...; //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref); //Explicitly bind and obtain a pointer to the local proxy
obj_ptr->do_something(); //Invoke a method on the local proxy
```
Parameter Passing

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

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 sending 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) A sends message and continues
A stopped running

B is not running
B starts and receives message

(b) A sends message and waits until accepted
Message is stored at B's location for later delivery

Accepted

Time

Persistent synchronous communication

(c) Transient asynchronous communication

A sends message and continues

Message can be sent only if B is running

B receives message

Time

(d) Transient synchronous communication

Send request and wait until received

Request is received

B Running, but doing something else

Process request

Time

Receipt-based transient synchronous communication

From: Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice Hall, Inc. 2002
Transient Messaging Alternatives (2)

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

Example: Socket primitives for TCP/IP.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication endpoint</td>
</tr>
<tr>
<td>Bind</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>Listen</td>
<td>Announce willingness to accept connections</td>
</tr>
<tr>
<td>Accept</td>
<td>Block caller until a connection request arrives</td>
</tr>
<tr>
<td>Connect</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>Send</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>Receive</td>
<td>Receive some data over the connection</td>
</tr>
<tr>
<td>Close</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

Connection-Oriented Communication Using Sockets

Persistent Messaging

- Usually called message-queueing 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:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block.</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue.</td>
</tr>
</tbody>
</table>
General Architecture of a Message-Queuing System (1)

- Look-up transport-level address of queue
- Queue-level address
- Address look-up database

General Architecture of a Message-Queuing System (2)

- The get
Message Brokers