Communication

Inter-process communication is at the heart of all distributed systems
Based on low-level message passing offered by the underlying network
Protocols: rules for communicating processes structured in layers
Four widely-used models:
- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)
- Message-Oriented Middleware (MOM)
- Streams

Topics to be covered

PART 1
- Layered Protocols
- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)

PART 2
- Message-Oriented Middleware (MOM)
- Streams

Layered Protocols

General Structure
Based on low-level message passing
A wants to communicate with B
A builds a message in its own address space
A executes a call to the OS to send the message
Need to agree on the meaning of the bits being sent

Layered Protocols

Processes define and adhere to rules (protocols) to communicate
Protocols are structured into layers - each layer deals with a specific aspect of communication
Each layer uses the services of the layer below it - an interface specifies the services provided by the lower layer to the upper layers
The upper layer sees the lower layer as a black box (benefit?)
Layered Protocols

Layer n on machine 1 talks with layer n on machine 1 based on the Layer n protocol.

Protocol suite or protocol stack: collection of protocols used in a particular system.
Each protocol adds a header.

The OSI Model

The ISO OSI or the OSI model
Designed to allow open systems to communicate.

Two general type of protocols:
• Connection-oriented: before exchanging data, the sender and the receiver must establish a connection (e.g., telephone), possibly negotiate the protocols to be used, release the connection when done.
• Connectionless: no setup in advance (e.g., sending an email).

The OSI Model

• Each layer provides an interface to the one above.
• Message send (downwards) Message received (upwards) example
• Each layer adds a header.

Low-level Layers

These layers implement the basic functions of a computer network.

Low-level Layers: The Physical Layer

Physical layer:
Concerns with transmitting 0s and 1s
Standardizing the electrical, mechanical, and signaling interfaces so that when A sends a 0 bit, it is received as a 0.

Issues:
• How many volts to use for 0 and 1.
• How many bits per sec (data rates).
• Whether to transmit in both directions (duplex/simpler).

Example standard: RS-232-C for serial communication lines.

The specification and implementation of bits, and their transmission between sender and receiver.
Low-level Layers: The Data Link Layer

Data link layer:
- Group bits into frames and sees that each frame is correctly received
- Puts a special bit pattern at the start and end of each frame (to mark them) as well as a checksum
- If checksums differ, requests a retransmission
- Frames are assigned sequence numbers

This layer prescribes the transmission of a series of bits into a frame to allow for error and flow control.

Low-level Layers: The Network Layer

Network layer:
- Deals with the fact that communication might require multi-hops
- At each hop, through which link to forward the packet
- Routing: choose the best ("delay-wise") path
- Example protocol at this layer: connectionless IP (part of the Internet protocol suite)
  - IP packets: each one is routed to its destination independent of all others.
  - No internal path is selected or remembered

This layer describes how packets in a network of computers are to be routed.

Low-level Layers

Physical Layer: specification and implementation of bits, transmission between sender and receiver
Data Link Layer: groups bits into frames, error and flow control
Network Layer: routes packets

NOTE

For many distributed systems, the lowest level interface is that of the network layer.

Transport Protocols

Turns the underlying network into something that an application developer can use

Transport Layer

Reliable connection
- The transport layer provides the actual communication facilities for most distributed systems
- Breaks a message received by the application layer into packets and assigns each one of them a sequence number and sends them all
- Header: which packets have been sent, received, there is room for, need to be retransmitted
- Reliable connection-oriented transport connections built on top of connection-oriented (all packets arrive in the correct sequence, if they arrive at all) or connectionless network services
Transport Layer

Standard (transport-layer) Internet protocols:
- Transmission Control Protocol (TCP): connection-oriented, reliable, stream-oriented communication (TCP/IP)
- Universal Datagram Protocol (UDP): connectionless, unreliable (best-effort) datagram communication (just IP with minor additions)

TCP vs UDP
Works reliably over any network
Considerable overhead
use UDP + additional error and flow control for a specific application

Higher-level Layers
In practice, only the application layer is used

Upper Layers
Session Layer
Maintain "logical" sessions using as many transport connections as necessary
Presentation Layer
Deals with non-uniform data representation (describing the messages in a platform-independent format and sending the descriptions along with data) and with compression and encryption

Application Layer
Intended to contain a collection of standard network applications, such as those for email, file transfer, etc.
From the OSI reference model, all distributed systems just applications
Many application protocols are directly implemented on top of transport protocols, doing a lot of application-independent work.

OSI vs TCP/IP Model

OSI reference model
TCP/IP not an official reference model (many details regarding the interfaces left open to implementation) - but the de facto Internet communication protocol
**Service Primitives**

- LISTEN: block waiting for an incoming connection
- CONNECT: establish a connection with a waiting host
- RECEIVE: block waiting for an incoming message
- SEND: send a message to a host
- DISCONNECT: terminate a connection

**Middleware Layer**

Middleware is invented to provide common services and protocols that can be used by many rich set of communication protocols, but which allow different applications to communicate:

- Marshaling and unmarshaling of data, necessary for integrated systems
- Naming protocols, so that different applications can easily share resources
- Security protocols, to allow different applications to communicate in a secure way
- Scaling mechanisms, such as support for replication and caching
- Authentication protocols, authorization
- Atomicity

**Middleware Protocols**

An adapted reference model for networked communication.

**Remote Procedure Call (RPC)**

**Basic idea:**

Allow programs to call procedures located on other machines.

**Some issues:**

- Calling and called procedures in different address spaces
- Parameter passing
- Crash of each machine

**Basic RPC Model**

Parameter Passing Variations

**Conventional Procedure Call**

Principle: "communication" with local procedure is handled by copying data to/from the stack (with a few exceptions)

Example: incr(i, i), (adds 1 to each parameter)

- Call-by-Value, i = 0
- Call-by-Reference, (push the address of the variable), i = 2
- Call-by-Copy/Restore

The value is copied in the stack as in call-by-value, and then copied back by the called procedure, i = 1
Client and Server Stubs
RPC supports location transparency (the calling procedure does not know that the called procedure is remote)

Client stub:
- local version of the called procedure
- called using the "stack" sequence
- it packs the parameters into a message and requests this message to be sent to the server (calls send)
- it calls receive and blocks till the reply comes back
When the message arrives, the server OS passes it to the server stub

Server Stub:
- typically waits on receive
- it transforms the request into a local procedure call
- after the call is completed, it packs the results, calls send
- it calls receive again and blocks

Steps of a Remote Procedure Call
1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client’s OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server’s OS sends message to client’s OS
9. Client’s OS gives message to client stub
10. Stub unpacks result, returns to client

Parameter Passing
Parameter marshaling: There is more than just wrapping parameters into a message:
- Client and server machines may have different data representations (think of byte ordering)
- Wrapping a parameter means transforming a value into a sequence of bytes
- Client and server have to agree on the same encoding:
  - How are basic data values represented (integers, floats, characters)
  - How are complex data values represented (arrays, unions)
- Client and server need to properly interpret messages, transforming them into machine-dependent representations.
Passing Reference Parameters

Pointer refers to the address space of the process it is being used

Solutions:

- Forbid pointers and reference parameters in general
- Use copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values (only Ada supports this model).

RPC assumes all data that is to be operated on is passed by parameters. Excludes passing references to (global) data.

One optimization, if the stubs know which are parameters are input and output parameters -> eliminate copying

What about pointers to complex (arbitrary) data structures?

Parameter Specification and Stub Generation

Need to agree on:

- Encoding rules (message format, representation of simple data structures)
- Actual exchange of messages (e.g., TCP/IP)
- Implement the stubs

Stubs for the same protocol and different procedures differ only in their interfaces to the applications

Interface Definition Language (IDL)

Extensions

- Calls to local procedures
- Asynchronous RPC

Doors

Try to use the RPC mechanism as the only mechanism for interprocess communication (IPC).

Doors are RPCs implemented for processes on the same machine: a single mechanism for communication (procedure calls, but with doors, it is not transparent)

Server calls `door_create`: registers a door, an id is returned

`fattach`: associates a symbolic name with the id

Client invokes a door using `door_call`, the id and any parameters

The OS does an upcall to the server

To return the result `door_return`

Asynchronous RPC

Try to get rid of the strict request-reply behavior, and let the client continue without waiting for an answer from the server.

Traditional RPC: the client waits for the reply

Asynchronous RPC: the server immediately sends a reply back to the client the moment the RPC request is received, after which it calls the requested procedure

Deferred Synchronous RPC: two asynchronous RPCs combined

The client uses asynchronous RPC to call the server

The server uses asynchronous RPC to send the reply

One way RPC: the client does not wait at all (reliability?)
Performing an RPC

At-most-one semantics: no call is ever carried out more than once, even in the case of system crashes

Idempotent remote procedure: a call may be repeated multiple times

DCE RPC

Let the developer concentrate on only the client- and server-specific code; let the RPC system (generators and libraries) do the rest.

Writing a Client and a Server

IDL permits procedure declarations (similar to function prototypes in C). Type definitions, constant declarations, etc. to provide information to correctly marshal/unmarshal parameters/results. Just the syntax (no semantics)

Global unique identifier

Generate a prototype IDL with a unique id

Edit the IDL, fill in the names of the remote procedures and their parameters

The steps in writing a client and a server in DCE RPC

Binding a Client to a Server

1. Locate the server machine
2. Locate the server on the machine; need to know an endpoint (port) on the server machine to which it can send messages

A table of (server, endpoints) is maintained on each server machine by a process called the DCE daemon

The server asks the OS for an endpoint and registers this endpoint with the DCE daemon

The client asks the DCE daemon at the server’s machine to lookup the endpoint

RPCgen

Check out the web page for an example

Programmer writes an example.x file with the definitions of remote procedures (their prototype) and other variables

RPCgen generates:

- example.h (header file, function prototypes)
- example_svc.c (server stub)
- example_clnt.c (client stub)
- example_client.c (template, the programmer edits this file, procedure calls)
- example_server.c (template, the programmer edits this file)

Remote Object Invocation

Distributed Objects

Remote Object Invocation

Parameter Passing
Distributed Objects

Expand the idea of RPCs to invocations on remote objects

- Data (state) and operations on those data encapsulated into an object
- Operations are implemented as methods and are accessible through interfaces
- An object offers only its interface to clients.
  An object may implement many interfaces:
  Given an interface definition, there may be several objects that offer an implementation for it
- An interface and its implementation on different machines

A client binds to a distributed object: an implementation of the object’s interface, called a proxy, is loaded into the client’s address space

Proxy (analog to a client stub):
Marshals method invocations into messages & Un-marshals reply messages
Actual object at a server machine: offers the same interface

Skeleton (analog to server stub):
Un-marshals requests to proper method invocations at the object’s interface of the server

Note: the object itself is not distributed, aka remote object

Compile-time objects:
Related to language-level objects (e.g., Java, C++)
Objects defined as instances of a class
Compiling the class definition results in code that allows to instantiate Java objects
Language-level objects, from which proxy and skeletons are automatically generated.
Depends on the particular language

Runtime objects:
Can be implemented in any language, but require use of an object adapter that makes the implementation appear as an object.
Adapter: objects defined based on their interfaces
Register an implementation at the adapter

Transient objects: live only by virtue of a server: if the server exits, so will the object.
Persistent objects: live independently from a server: if a server exits, the object’s state and code remain (passively) on disk

Binding a Client to an Object

Provide system-wide object references, freely passed between processes on different machines.
Reference denotes the server machine plus an endpoint for the object server, an id of which object

When a process holds an object reference, it must first bind to the object

Bind: the local proxy (stub) is instantiated and initialized for specific object – implementing an interface for the object methods

Two ways of binding:
Implicit binding: Invoke methods directly on the referenced object (requires global references)
Explicit binding: Client must first explicitly bind to object before invoking it (generally returns a pointer to a proxy that then becomes locally available)
Basic RMI

Assume client stub and server skeleton are in place
- Client invokes method at stub
- Stub marshals request and send it to server
- Server ensures referenced object is active
  - Created separate process to hold object
  - Load the object into server process
- Request is unmarshalled by object's skeleton, and referenced object is invoked
  - If request contained an object reference, invocation is applied recursively
  - Result is marshalled and passed back to client
- Client stub unmarshals reply and passes result to client application

Static vs Dynamic RMI

Remote Method Invocation (RMI)
- Static invocation: the interfaces of an object are known when the client application is being developed
  If interfaces change, the client application must be recompiled
- Dynamic invocation: the application selects at runtime which method it will invoke at a remote object
  `invoke(object, method, input_parameters, output_parameters)`
  `method` is a parameter, `input_parameters`, `output_parameters` data structures
  Static: `object.append(int)`
  Dynamic: `invoke(object, id(append), int)`
  `id(append)` returns an id for the method append
  Example uses: browsers, batch processing service to handle invocation requests

Object References as Parameters
When invoking a method with an object reference as a parameter, when it refers to a remote object, the reference is copied and passed as a value parameter (pass-by-reference)
When the reference refers to a local object (i.e., an object in the same address space as the client) the referred object is copied as a whole and passed along with the invocation (pass-by-value)

Java RMI
- Distributed objects integrated into the language
- Remote objects (i.e., state on a single machine, interfaces available to many) the only form of distributed objects
- Interfaces implemented by proxies that appear as a local object
- Differences between remote and local objects (violating distribution transparency)
  - Cloning
    - Cloning a local object O results in a new object of the same type as O and with exactly the same state
    - Cloning of a remote object O executed only by the server - proxies of the actual object are not cloned (how to bind to the clone to access it)

Java RMI
- Any serializable object type can be used as a parameter to an RMI
- A type is serializable if it can be marshalled
- Local objects are passed by value; whereas remote objects are passed by reference
- A remote object is built from two different classes:
  - server class: implementation of the server-side code
  - client class: implementation of the proxy (needs the server's network address and endpoint)
- Proxies are serializable, thus can be marshalled and passed as parameters
  (sent over to other processes, which can unmarshal them and use them as references to remote objects)
Java RMI

Check out the web page for an implementation