Addressing applications (transport layer)

Communication endpoint at the machine
- **Port number**: 16-bit value
- Port number – transport endpoint
  - Identifies a specific data stream
- Some services use well-known port numbers (0 – 1023)
  - IANA: Internet Assigned Numbers Authority (www.iana.org)
  - Also see the file /etc/services
    - ftp: 21/TCP
    - ssh: 22/tcp
    - smtp: 25/tcp
    - http: 80/tcp
    - ntp: 123/udp
- Ports for proprietary apps: 1024 – 49151
- Dynamic/private ports: 49152 – 65535
- To communicate with applications, we use a transport layer protocol and an IP address and port number

Network API

- App developers need access to the network
- A **Network Application Programming Interface (API)** provides this
- Core services provided by the operating system
  - Operating System controls access to resources
- Libraries may handle the rest
  - We will only look at IP-based communication

Programming: connection-oriented protocols

1. **establish connection**
   - dial phone number
2. **[negotiate protocol]**
   - decide on a language
3. **exchange data**
4. **terminate connection**
   - hang up

**virtual circuit service**
- provides illusion of having a dedicated circuit
- messages guaranteed to arrive in-order
- application does not have to address each message

**Not to be confused with virtual circuit networks**
- Which provide constant latency & guaranteed bandwidth
- TCP simulates a virtual circuit network ... sort of

Programming: connectionless protocols

**datagram service**
- client is not positive whether message arrived at destination
- no state has to be maintained at client or server
- cheaper but less reliable than virtual circuit service

**analogous to mailbox**
- no call setup
- send/receive data
  - (each packet addressed)
- no termination
- drop letter in mailbox
  - (each letter addressed)
Sockets

- Dominant API for transport layer connectivity
- Created at UC Berkeley for 4.2BSD Unix (1983)
- Design goals
  - Communication between processes should not depend on whether they are on the same machine
  - Communication should be efficient
  - Interface should be compatible with files
  - Support different protocols and naming conventions
- Sockets is not just for the Internet Protocol family

What is a socket?

Abstract object from which messages are sent and received
- Looks like a file descriptor
- Application can select particular style of communication
  - Virtual circuit (connection-oriented), datagram (connectionless), message-based, in-order delivery
- Unrelated processes should be able to locate communication endpoints
  - Sockets can have a name
  - Name should be meaningful in the communications domain
  - E.g., Address & port for IP communications

Connection-Oriented (TCP) socket operations

Client

Create a socket
Name the socket
Set the socket for listening
Connect to the other side
read / write byte streams
close the socket

Server

Create a socket
Name the socket
Set the socket for listening
Wait for and accept a connection: get a socket for the connection
read / write byte streams
close the socket

Connectionless (UDP) socket operations

Client

Create a socket
Name the socket
Send a message
Receive a message
close the socket

Server

Create a socket
Name the socket
Send a message
Receive a message
close the socket

Java provides shortcuts that combine calls

Example

```
import java.net.*;
import java.io.*;

Socket s = new Socket("www.rutgers.edu", 2211);
```

Python Example

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
s.sendall(message)
```

Java

```
Creating a socket

s = new Socket("www.rutgers.edu", 2211);
```

C

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

Note: try/except blocks are missing

Connectionless (UDP) socket operations

Client

Create a socket
Name the socket
Send a message
Receive a message
close the socket

Server

Create a socket
Name the socket
Send a message
Receive a message
close the socket
Socket-based communication

- Socket API: all we get from the OS to access the network
- Socket = distinct end-to-end communication channels

Read/write model
- Send a bunch of bytes
- Read a bunch of bytes
- Send a bunch of bytes
- Read a bunch of bytes
- ...

Application implements its protocol
- Line-oriented, text-based protocols common
  - Not always efficient but easy to debug & use

Sample SMTP Interaction

SMTP = Simple Mail Transfer Protocol

Sample SMTP Interaction

Regular procedure calls

You write:

```c
x = f(a, "test", 5);
```

The compiler parses this and generates code to:

- Push the value 5 on the stack
- Push the address of the string "test" on the stack
- Push the current value of a on the stack
- Generate a call to the function f

In compiling f, the compiler generates code to:

- Push registers that will be clobbered on the stack to save the values
- Adjust the stack to make room for local and temporary variables
- Before a return, unadjust the stack, put the return data in a register, and issue a return instruction
Implementing RPC

No architectural support for remote procedure calls

Simulate it with tools we have (local procedure calls)

Simulation makes RPC a language-level construct

The OS gives us sockets

The compiler creates code to send messages to invoke remote functions

The trick:

Create **stub functions**

to make it appear to the user that the call is local

On the client

The stub function has the function’s interface

Packages parameters and calls the server

On the server

The stub function (skeleton) receives the request and calls the local function

**1. Client calls stub (params on stack)**

- Client functions
- Client stub (proxy)
- Network routines

**2. Stub marshals params to net message**

- Client functions
- Client stub (proxy)
- Network routines

Marshalling = put data in a form suitable for transmission over a network (serialized)

**3. Network message sent to server**

- Client functions
- Client stub (proxy)
- Network routines

**4. Receive message: send it to server stub**

- Client functions
- Client stub (proxy)
- Network routines

- Server functions
- Server stub (skeleton)
- Network routines
5. Unmarshal parameters, call server function

6. Return from server function

7. Marshal return value and send message

8. Transfer message over network

9. Receive message: client stub is receiver

10. Unmarshal return value, return to client code
A proxy looks like the remote function
• Client stub has the same interface as the remote function
• Looks & feels like the remote function to the programmer
  – But its function is to
    • Marshal parameters
    • Send the message
    • Wait for a response from the server
    • Unmarshal the response & return the appropriate data
    • Generate exceptions if problems arise

A skeleton is really two parts
• Dispatcher
  – Receives client requests
  – Identifies appropriate function (method)
• Skeleton
  – Unmarshals parameters
  – Calls the local server procedure
  – Marshals the response & sends it back to the dispatcher
• Invisible to the programmer
  – The programmer doesn’t deal with any of this
  – Dispatcher + Skeleton may be integrated
    • Depends on implementation

RPC Benefits
• RPC gives us a procedure call interface
• Writing applications is simplified
  – RPC hides all network code into stub functions
  – Application programmers don’t have to worry about details
    • Sockets, port numbers, byte ordering
• Where is RPC in the OSI model?
  – Layer 5: Session layer: Connection management
  – Layer 6: Presentation: Marshaling/data representation
  – Uses the transport layer (4) for communication (TCP/UDP)

RPC has challenges

Parameter passing
Pass by value
  – Easy: just copy data to network message
Pass by reference
  – Makes no sense without shared memory

Pass by reference?
1. Copy items referenced to message buffer
2. Ship them over
3. Unmarshal data at server
4. Pass local pointer to server stub function
5. Send new values back

To support complex structures
  – Copy structure into pointerless representation
  – Transmit
  – Reconstruct structure with local pointers on server
Representing data

No such thing as *incompatibility problems* on local system

Remote machine may have:
- Different byte ordering
- Different sizes of integers and other types
- Different floating point representations
- Different character sets
- Alignment requirements

IP (headers) forced all to use **big endian** byte ordering for 16- and 32-bit values

Big endian: Most significant byte in low memory
  - SPARC V9, V8, Motorola 68000, older PowerPC

Little endian: Most significant byte in high memory
  - Intel/AMD IA-32, x64

Bi-endian: Processor may operate in either mode
  - ARM, PowerPC, MIPS, SPARC V9, IA-64 (Intel Itanium)

```c
main() {
    unsigned int n;
    char *a = (char *)&n;
    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x\n",
           a[0], a[1], a[2], a[3]);
}
```

Output on an Intel: 44, 33, 22, 11
Output on a PowerPC: 11, 22, 33, 44

Representing data: serialization

Need standard encoding to enable communication between heterogeneous systems

- **Serialization**
  - Convert data into a pointerless format: an array of bytes

- **Examples**
  - XDR (eXternal Data Representation), used by ONC RPC
  - JSON (JavaScript Object Notation)
  - W3C XML Schema Language
  - ASN.1 (ISO Abstract Syntax Notation)
  - Google Protocol Buffers

Representing data

**Implicit typing**
- only values are transmitted, not data types or parameter info
- e.g., ONC XDR (RFC 4506)

**Explicit typing**
- Type is transmitted with each value
- e.g., ISO's ASN.1, XML, protocol buffers, JSON

Where to bind?

Need to locate host and correct server process

Maintain a centralized DB that can locate a host that provides a particular service
(Birrell & Nelson’s 1984 proposal)

Challenges:
- Who administers this?
- What is the scope of administration?
- What if the same services run on different machines (e.g., file systems)?
Where to bind? – Solution 2

A server on each host maintains a DB of locally provided services.

Transport protocol

TCP or UDP? Which one should we use?

- Some implementations may offer only one (e.g. TCP)
- Most support several
  - Allow programmer (or end user) to choose at runtime

When things go wrong

- Local procedure calls do not fail
  - If they core dump, entire process dies
- More opportunities for error with RPC
- Transparency breaks here
  - Applications should be prepared to deal with RPC failure

When things go wrong

- Semantics of remote procedure calls
  - Local procedure call: exactly once
- A remote procedure call may be called:
  - 0 times: server crashed or server process died before executing server code
  - 1 time: everything worked well, as expected
  - 1 or more times: excess latency or lost reply from server and client retransmission

RPC semantics

- Most RPC systems will offer either:
  - at least once semantics
  - or at most once semantics
- Understand application:
  - idempotent functions: may be run any number of times without harm
  - non-idempotent functions: those with side-effects
- Try to design your application to be idempotent
  - Not always easy!
  - Store transaction IDs, previous return data, etc.

More issues

Performance

- RPC is slower … a lot slower (why?)

Security

- messages may be visible over network – do we need to hide them?
  - Authenticate client?
  - Authenticate server?
Programming with RPC

Language support
- Many programming languages have no language-level concept of remote procedure calls
  (C, C++, Java <=J2SE 5.0, …)
  • These compilers will not automatically generate client and server stubs
- Some languages have support that enables RPC
  (Java, Python, Haskell, Go, Erlang)
  • But we may need to deal with heterogeneous environments (e.g., Java communicating via XML)

Common solution
- Interface Definition Language (IDL): describes remote procedures
- Separate compiler that generate stubs (pre-compiler)

Interface Definition Language (IDL)

- Allow programmer to specify remote procedure interfaces
  (names, parameters, return values)
- Pre-compiler can use this to generate client and server stubs
  • Marshaling code
  • Unmarshaling code
  • Network transport routines
  • Conform to defined interface
- An IDL looks similar to function prototypes

RPC compiler

RPC API

What kind of services does an RPC system need?
- Name service operations
  - Export/lookup of binding information (ports, machines)
  - Support dynamic ports
- Binding operations
  - Establish client/server communications using appropriate protocol
    (establish endpoints)
- Endpoint operations
  - Listen for requests, export endpoint to name server
    (often the main program on the server)

RPC API

What kind of services does an RPC system need?
- Security operations
  - Authenticate client/server
- Internationalization operations (maybe)
- Marshaling/data conversion operations
- Stub memory management
  - Dealing with “reference” data, temporary buffers
- Program ID operations
  - Allow applications to access IDs of RPC interfaces
  - Can you pass references to remote functions to other processes?
The End