Multiple Access Protocols

- Share a communication medium
- Statistical multiplexing
  - No timeslots
  - Anyone can transmit when ready
  - But be prepared for collisions or dropped packets

The Internet

- Internet
  - Logical network on top of multiple physical networks
  - Packets routed through routers: store & forward
  - Fault tolerant
    - No central point of control
    - Packets can take alternate routes
    - Best effort delivery: no guarantees

  - End-to-end principle → smart endpoints, dumb network
    - Only network endpoints (hosts) should store state
    - Any application-specific functions should be implemented at the hosts
      - end-to-end vs. hop-by-hop
        - E.g., TCP state is preserved even if traffic moves to different routers

IP vs. OSI protocol stack

<table>
<thead>
<tr>
<th>OSI Layer</th>
<th>Internet Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>Middleware</td>
<td></td>
</tr>
<tr>
<td>Transport</td>
<td>TCP, UDP</td>
</tr>
<tr>
<td>Network</td>
<td>IP, IPv6</td>
</tr>
<tr>
<td>Data Link</td>
<td>Ethernet MAC</td>
</tr>
<tr>
<td>Physical</td>
<td>Internet protocol stack</td>
</tr>
</tbody>
</table>

IP transport layer protocols

Enable applications to communicate

- TCP: Transmission Control Protocol
  - Connection-oriented service – operating system keeps state
  - Full-duplex connection: both sides can send messages over the same link
  - Reliable data transfer: the protocol handles retransmission
  - In-order data transfer: the protocol keeps track of sequence numbers
  - Flow control: receiver stops sender from sending too much data
  - Congestion control: “plays nice” on the network

- UDP: User Datagram Protocol
  - Connectionless service: lightweight transport layer over IP
  - Data may be lost
  - Data may arrive out of sequence
  - Checksum for corrupt data: operating system drops bad packets
  - Sender can keep blasting data to the receiver even if it can’t keep up
Point-to-Point Communication

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

Connection-Oriented (TCP) socket operations

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a socket</td>
<td>socket</td>
</tr>
<tr>
<td>Name the socket (assign local address, port)</td>
<td>socket</td>
</tr>
<tr>
<td>Connect to the other side</td>
<td>accept</td>
</tr>
<tr>
<td>read/write byte streams</td>
<td>read/write byte streams</td>
</tr>
<tr>
<td>close the socket</td>
<td>close</td>
</tr>
<tr>
<td>close the listening socket</td>
<td>close</td>
</tr>
</tbody>
</table>

Connectionless (UDP) socket operations

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a socket</td>
<td>socket</td>
</tr>
<tr>
<td>Name the socket (assign local address, port)</td>
<td>socket</td>
</tr>
<tr>
<td>Send a message</td>
<td>sendto</td>
</tr>
<tr>
<td>Receive a message</td>
<td>recvfrom</td>
</tr>
<tr>
<td>close the socket</td>
<td>close</td>
</tr>
</tbody>
</table>

Java provides shortcuts that combine calls

```java
String host = "www.rutgers.edu";
byte[] content = "This is the message body.
Headers may define the structure of the message but are ignored for delivery.
Hi, This is a test
250 2.0.0 allFGE2S13883 Message accepted for delivery
221 2.0.0 aramis.rutgers.edu closing connection";
...
```

Sample SMTP Interaction

SMTP = Simple Mail Transfer Protocol

```
Sending to python@r1.pin0ost.com... 200 2.0.0 Add recipient python@r1.pin0ost.com
250 2.0.0 Local Name Check OK for python@r1.pin0ost.com
250 2.0.0 Command Accepted
SMTP = Simple Mail Transfer Protocol
```
TCP Sockets: Disambiguating Messages

Remote Procedure Calls

Problems with the sockets API

RPC

Regular procedure calls

Implementing RPC

TCP Sockets: Disambiguating Messages

Remote Procedure Calls

Problems with the sockets API

RPC

Regular procedure calls

Implementing RPC
Implementing RPC

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

server functions

client stub (proxy)

server stub (skeleton)

network routines

---

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

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

client functions

server functions

client stub (proxy)

server stub (skeleton)

network routines

---

**3. Network message sent to server**

Client functions

server functions

client stub (proxy)

server stub (skeleton)

network routines

---

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

Client functions

server functions

client stub (proxy)

server stub (skeleton)

network routines

---

**5. Unmarshal parameters, call server function**

Client functions

server functions

client stub (proxy)

server stub (skeleton)

network routines
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
Representing data

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, Motorola 680x0, 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)

```
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

Where to bind?

Need to locate host and correct server process

- **Solution 1**
  - Maintain a centralized DB that can locate a host that provides a particular service
  - (Birrell & Nelson’s 1984 proposal)

  Challenges:
  - Who administrators this?
  - What is the scope of administration?
  - What if the same services run on different machines (e.g., file systems)?

- **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
  - *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

- IDL → client code (main)
- client stub
- compiler
- client
- Data conv.
- compiler
- server
- headers
- server functions
- server skeleton

Writing the program

- Client code has to be modified
  - Initialize RPC-related options
    - Identify transport type
    - Locate server/service
    - Handle failure of remote procedure calls
- Server functions
  - Generally need little or no modification

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