Some terminology

- A **program** is the code you type in
- A **process** is what you get when you run it
- A **message** is used to communicate between processes. Arbitrary size.
- A **packet** is a fragment of a message that might travel on the wire. Variable size but limited, usually to 1400 bytes or less.
- A **protocol** is an algorithm by which processes cooperate to do something using message exchanges.

More terminology

- A **network** is the infrastructure that links the computers, workstations, terminals, servers, etc.
  - It consists of routers
  - They are connected by communication links
- A **network application** is one that fetches needed data from servers over the network
- A **distributed system** is a more complex application designed to run on a network. Such a system has multiple processes that cooperate to do something.

More terminology

- A **real-world network** is what we work on. It has computers, links that can fail, and some problems synchronizing time. But this is hard to model in a formal way.
- An **asynchronous distributed system** is a theoretical model of a network with no notion of time
- A **synchronous distributed system**, in contrast, has perfect clocks and bounds all events, like message passing.

Model we’ll use?

- Our focus is on real-world networks, halting failures, and extremely practical techniques
- The closest model is the asynchronous one; we use it to reason about protocols
  - Most often, employ asynchronous model to illustrate techniques we can actually implement in real-world settings
  - And usually employ the synchronous model to obtain impossibility results
- **Question:** why not prove impossibility results in an asynchronous model, or use the synchronous one to illustrate techniques that we might really use?
Example: Server replication

- Suppose that our Air Traffic Control needs a highly available server.
- One option: “primary/backup”
  - We run two servers on separate platforms
  - The primary sends a log to the backup
  - If primary crashes, the backup soon catches up and can take over

### Split brain Syndrome...

Clients initially connected to primary, which keeps backup up to date. Backup collects the log

### Split brain Syndrome...

Clients initially connected to primary, which keeps backup up to date. Backup collects the log

### Split brain Syndrome...

Clients initially connected to primary, which keeps backup up to date. Backup collects the log

### Split brain Syndrome...

Clients initially connected to primary, which keeps backup up to date. Backup collects the log

### Implication?

- Air Traffic System with a split brain could malfunction disastrously!
  - For example, suppose the service is used to answer the question “is anyone flying in such-and-such a sector of the sky”
  - With the split-brain version, each half might say “nope”... in response to different queries!
- Another example: duplicate train tickets

### Can we fix this problem?

- The essential insight is that we need some form of “agreement” on which machines are up and which have crashed
- Can’t implement “agreement” on a purely 1-to-1 (hence, end-to-end) basis.
  - Separate decisions can always lead to inconsistency
  - So we need a “membership service”...
Can we fix this problem?

- Yes, many options, once we accept this
  - Just use a single server and wait for it to restart
  - This common today, but too slow for ATC
  - Give backup a way to physically "kill" the primary, e.g., unplug it
  - If backup takes over... primary shuts down
  - Or require some form of "majority vote"
  - As mentioned, maintains agreement on system status

Bottom line? You need to anticipate the issue... and to implement a solution.

Definition of a Distributed System (1)

- A distributed system is a collection of independent computers that appears to its users as a single coherent system. [Tanenbaum, 2002]

Definition of a Distributed System (2)

A distributed system organized as middleware. Note that the middleware layer extends over multiple machines.

Interaction between a client and server

Outline

- Terminology
- Client Server Model
- OSI Model vs. Middleware Model
- Summary

Berkeley Sockets (1)

<table>
<thead>
<tr>
<th>Primitve</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>

Socket primitives for TCP/IP.
Berkeley Sockets (2)

Connection-oriented communication pattern using sockets.

An Example Client and Server (1)

The `header.h` file used by the client and server.

An Example Client and Server (2)

A sample server.

An Example Client and Server (3)

A client using the server to copy a file.

Organize a search engine into three layers

Multitiered Architectures (1)

Alternative client-server organizations (a) – (e).
Multitiered Architectures (2)

An example of a server acting as a client.

Modern Architectures

An example of horizontal distribution of a Web service.

Outline

- Terminology
- Client-Server Model
- OSI Model vs. Middleware Model
- Summary

Classic view of network API

- Start with host name
- Get an IP address
- Make a socket (protocol, address)

Classic view of network API

- Start with host name
- Get an IP address
- Make a socket (protocol, address)
Classic view of network API

- Start with host name
- Get an IP address
- Make a socket (protocol, address)
- Send byte stream (TCP) or packets (UDP)

Classic approach “broken” in many ways

- IP address different depending on who asks for it
- Address may be changed in the network
- IP address may not be reachable (even though destination is up and attached)
- Or may be reachable by you but not another host
- IP address may change in a few minutes or hours
- Packets may not come from who you think they come from (network caches)

Open Systems & Protocols

- An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalized in what are called protocols.
- A distinction is made between two general types of protocols.
  - Connection-oriented protocols, e.g., telephone
  - Connectionless protocols, e.g., dropping a letter in a mailbox

OSI protocol layers: Oft-cited Standard

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>The program using a communication connection</td>
</tr>
<tr>
<td>Presentation</td>
<td>Software to encode data into messages, and decode on reception</td>
</tr>
<tr>
<td>Session</td>
<td>Logic associated with guaranteeing end-to-end reliability and flow control, if desired</td>
</tr>
<tr>
<td>Transport</td>
<td>Software for fragmenting big messages into small packets</td>
</tr>
<tr>
<td>Network</td>
<td>Routing functionality, limited to small packets</td>
</tr>
<tr>
<td>Data-link</td>
<td>The protocol that represents packets on the wire</td>
</tr>
<tr>
<td>Hardware</td>
<td>Hardware for representing bits on the wire</td>
</tr>
</tbody>
</table>

- OSI is tied to a TCP-style of connection
- Match with modern protocols is poor
- We are mostly at “layer 4” – session

Layered Protocols (1)

Layers, interfaces, and protocols in the OSI model.

Layered Protocols (2)

- Communications stack consists of a set of services, each providing a service to the layer above, and using services of the layer below
  - Each service has a programming API, just like any software module
  - Each service has to convey information one or more peers across the network
  - This information is contained in a header
  - The headers are transmitted in the same order as the layered services
A typical message as it appears on the network:

- **Layered Protocols (3)**

  ![Diagram of layered protocols](image)

  - **Data link layer header**
  - **Network layer header**
  - **Transport layer header**
  - **Session layer header**
  - **Presentation layer header**
  - **Application layer header**

  Bits that actually appear on the network:

- **Protocol layering example**

  ![Diagram of protocol layering example](image)

  **Browser process**

  - HTTP
  - TCP
  - IP
  - Link1
  - Link1
  - IP
  - Link2
  - Link2
  - IP
  - Physical Link 1
  - Physical Link 2

  **Web server process**

  - HTTP
  - TCP
  - IP
  - Link1
  - Link1
  - IP
  - Link2
  - Link2
  - IP
  - Physical Link 1
  - Physical Link 2

  **Browser wants to request a page.** Calls HTTP with the web address (URL). HTTP’s job is to convey the URL to the web server. HTTP learns the IP address of the web server, adds its header, and calls TCP.

- **Protocol layering example**

  ![Diagram of protocol layering example](image)

  **TCP’s job is to work with server to make sure bytes arrive reliably and in order.** TCP adds its header and calls IP. (Before that, TCP establishes a connection with its peer.)

- **IP’s job is to get the packet routed to the peer through zero or more routers.** IP determines the next hop from the destination IP address. IP adds its header and calls the link layer (i.e., Ethernet) with the next hop address.

- **The link’s job is to get the packet to the next physical box (here a router).** It adds its header and sends the resulting packet over the “wire”.

- **Router**

  - HTTP
  - TCP
  - IP
  - Link1
  - Link1
  - IP
  - Link2
  - Link2
  - IP
  - Physical Link 1
  - Physical Link 2
Protocol layering example

The router’s link layer receives the packet, strips the link header, and hands the result to the IP forwarding process.

The router’s IP forwarding process looks at the destination IP address, determines what the next hop is, and hands the packet to the appropriate link layer with the appropriate next hop link address.

The packet goes over the link to the web server, after which each layer processes and strips its corresponding header.

Discussion between a receiver and a sender in the data link layer.

Network Layer

- How to choose the best path is called routing.
  - The shortest route is not always the best one.
  - The amount of delay on a given route
  - The amount of traffic and messages queued up
  - The day can thus change over the course of time
  - Some algorithms try to adapt to changing loads.
  - Others are content to make decisions based on long-term averages.
- Internet Protocol (IP) is part of the Internet protocol suite
  - An IP packet can be sent without any setup
  - Each IP packet is routed to its destination independent of all others
Remote Procedure Call

- Allows a program to cause a procedure to execute in another address space.
- The programmer would write essentially the same code whether the subroutine is local to the executing program, or remote.
- When the software in question is written using object-oriented principles, RPC may be referred to as remote invocation or remote method invocation.
- RPC is an easy and popular paradigm for implementing the client-server model of distributed computing.

Conventional Procedure Call

(a) Parameter passing in a local procedure call: the stack before the call to read.
(b) The stack while the called procedure is active
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

RPC Failure

- More failure modes than simple procedure calls
- Machine failures
- Communication failures
- RPCs can return "failure" instead of results
- What are possible outcomes of failure?
  - Procedure did not execute
  - Procedure executed once
  - Procedure executed multiple times
  - Procedure partially executed
- Generally desired semantics: at most once

Implementing at most once semantics

- Danger: Request message lost
  - Client must retransmit requests when it gets no reply
- Danger: Reply message may be lost
  - Client may retransmit previously executed request
  - Okay if operations are idempotent, but many are not
  - E.g., process order, charge customer, ...
  - Server must keep "replay cache" to reply to already executed requests
- Danger: Server takes too long to execute procedure
  - Client will retransmit request already in progress
  - Server must recognize duplicate—can reply "in progress"

Server crashes

- Danger: Server crashes and reply lost
  - Can make replay cache persistent—slow
- Danger: Server crashes during execution
  - Can log enough to restart partial execution—slow and hard
  - Can hope reboot takes long enough for all clients to fail
- Can use "cookies" to inform clients of crashes
  - Server gives client cookie which is time of boot
  - Client includes cookie with RPC
  - After server crash, server will reject invalid cookie

Passing Value Parameters (1)

Steps involved in doing remote computation through RPC

Passing Value Parameters (2)

<table>
<thead>
<tr>
<th>Original message on the Pentium</th>
<th>The message after receipt on the SPARC</th>
<th>The message after being inverted. The little numbers in boxes indicate the address of each byte</th>
</tr>
</thead>
</table>
Parameter Specification and Stub Generation

- A procedure
- The corresponding message.

```
foo_bar(char x, float y, int z){
    ... 
}
```

```
interface foo_bar{
    x y z[];
}
```

Interface Definition Language (1)

- IDL is a language used to describe a software component’s interface.
  - Describe an interface in a language-neutral way
  - enabling communication between software components that do not share a language
  - for example, between components written in C++ and components written in Java.

- Software systems based on IDLs include
  - Sun’s ONC RPC,
  - The Open Group’s Distributed Computing Environment,
  - IBM’s System Object Model,
  - the Object Management Group’s CORBA,
  - Facebook’s Thrift
  - and SOAP for Web services.

Interface Definition Language (2)

- Idea: Specify RPC call and return types in IDL
  - Compile interface description with IDL compiler.
- Output:
  - Native language types (e.g., C/java/C++ structs/classes)
  - Code to marshal (serialize) native types into byte streams
  - Stub routines on client to forward requests to server
- Stub routines handle communication details
  - Helps maintain RPC transparency, but
  - Still have to bind client to a particular server
  - Still need to worry about failures

Intro to SUN RPC

- Simple, no-frills, widely-used RPC standard
  - Does not emulate pointer passing or distributed objects
  - Programs and procedures simply referenced by numbers
  - Client must know server—no automatic location
  - Portmap service maps program #s to TCP/UDP port #s

- IDL: XDR – eXternal Data Representation
  - Compilers for multiple languages (C, java, C++)

Transport layer

- Transport layer transmits delimited RPC messages
  - In theory, RPC is transport-independent
  - In practice, RPC library must know certain properties
    - e.g., Is transport connected? Is it reliable?
- UDP transport: unconnected, unreliable
  - Sends one UDP packet for each RPC request/response
  - Each message has its own destination address
  - Server needs replay cache
- TCP transport (simplified): connected, reliable
  - Each message in stream prefixed by length
  - RPC library does not retransmit or keep replay cache

Sun XDR

- “External Data Representation”
  - Describes argument and result types:
    - struct message {
      int opcode;
      opaque cookie[8];
      string name<255>;
    }
  - Types can be passed across the network
- Sun rpcgen compiles to C
- Libasync rpc compiles to C++
  - Converts messages to native data structures
  - Generates marshaling routines (structs/byte stream)
  - Generates info for stub routines
Writing a Client and a Server

The steps in writing a client and a server in SUN RPC.

Binding a Client to a Server

Client-to-server binding in SUN RPC.

An Example of SUN RPC (1)

/* *
* @file avg.x
* @brief The average procedure receives an array of real numbers and returns the average of their values. This service handles a maximum of 200 numbers.
*/

const MAXAVGSIZE = 200;

struct input_data {
    double input_data<200>;
};

typeal struct input_data input_data;

program AVERAGEPROG {
    version AVERAGEVERS {
        double AVERAGE(input_data) = 1;
    } = 1;
} = 22855;

An Example of SUN RPC (2)

rpcgen avg.x

avg.h: function prototypes and data declarations needed for the application

avg_clnt.c: the stub program for our client (caller) process

avg_svc.c: the main program for our server (callee) process

avg_xdr.c: the XDR routines used by both the client and the server

Server Code for the Example

// @file avg_proc.c
#include <rpc/rpc.h>
#include "avg.h"
#include <stdio.h>

static double sum_avg;

double * average_1(input_data *input, CLIENT *client) {
    double *dp = input->input_data.input_data_val;
    u_int i;
    sum_avg = 0;
    for(i=1;i<=input->input_data.input_data_len;i++) {
        sum_avg = sum_avg + *dp; dp++;
    }
    sum_avg = sum_avg / input->input_data.input_data_len;
    return(&sum_avg);
}

double * average_1_svc(input_data *input, struct svc_req *svc) {
    CLIENT *client;
    return(average_1(input,client));
}

Client Code for the Example

// @file Ravg.c
#include "avg.h"
#include <stdlib.h>

void averageprog_1( char* host, int argc, char *argv[])
{
    CLIENT *clnt;
    double  *result_1, *dp, f;
    char *endptr;
    int i;
    input_data  average_1_arg;
    average_1_arg.input_data.input_data_val =
        (double*) malloc(MAXAVGSIZE*sizeof(double));
    dp = average_1_arg.input_data.input_data_val;
    for (i=1;i<=(argc - 2);i++) {
        f = strtod(argv[i+1],&endptr);
        printf("value = %e\n",f);
        *dp = f;
        dp++;
    }
    clnt = clnt_create(host, AVERAGEPROG,
        AVERAGEVERS, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror(host);   exit(1);
    }
    result_1 = average_1(&average_1_arg, clnt);
    if (result_1 == NULL) {
        clnt_perror(clnt, "call failed:");
    }
    clnt_destroy( clnt );
    printf("average = %e\n",*result_1);
}

main( int argc, char* argv[] )
{
    char *host;
    if(argc < 3) {
        printf( "usage: %s server_host value ...
",  argv[0]);
        exit(1);
    }
    if(argc > MAXAVGSIZE + 2) {
        printf("Two many input values\n");
        exit(2);
    }
    host = argv[1];
    averageprog_1( host, argc, argv);
}
Makefile for the Example

```
# @file Makefile
BIN = ravg avg_svc
GEN = avg_clnt.c avg_svc.c avg_xdr.c avg.h
RPCCOM = rpcgen
all: $(BIN)

ravg: ravg.o avg_clnt.o avg_xdr.o
$(CC) -o $@ ravg.o avg_clnt.o avg_xdr.o

ravg.o: ravg.c avg.h
$(CC) -g ravg.c -c

avg_svc: avg_proc.o avg_svc.o avg_xdr.o
$(CC) -o $@ avg_proc.o avg_svc.o avg_xdr.o

avg_proc.o: avg_proc.c avg.h

$(GEN): avg.x
$(RPCCOM) avg.x

clean cleanup:
rm -f $(GEN) *.o $(BIN)
```

Testing and Debugging the Application

```
./avg_svc
./usr/sbin/rpcinfo -p localhost

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Protocol</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>portmapper</td>
<td>2</td>
<td>tcp</td>
<td>111</td>
</tr>
<tr>
<td>portmapper</td>
<td>2</td>
<td>udp</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>udp</td>
<td>35368</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>tcp</td>
<td>37058</td>
</tr>
</tbody>
</table>

./ravg localhost $RANDOM $RANDOM $RANDOM
```

Testing and Debugging the Application

```
value = 9.196000e+03
value = 2.871200e+04
value = 3.198900e+04
average = 2.329900e+04
```

Extended RPC Models

- **Doors.** A door is a generic name for a procedure in the address space of a server process that can be called by processes colocated with the server.
- **Asynchronous RPCs,** a client immediately continues after issuing the RPC request:
  - The server immediately sends a reply back to the client the moment the RPC request is received.
  - After which it calls the requested procedure.
  - The client will continue without further blocking as soon as it has received the server's acknowledgement.

Doors

The principle of using doors as IPC mechanism.

Asynchronous RPC Models

1. **Asynchronous RPC (1):**
   - The interconnection between client and server in a traditional RPC.
   - The interaction using asynchronous RPC.

2. **Asynchronous RPC (2):**
   - A client and server interacting through two asynchronous RPCs.
Summary

- Various definitions of distributed systems have been given in the literature.
  - The definition has two aspects.
    - One deals with hardware.
    - The second one deals with software.
- TCP, UDP, IP provide a nice set of basic tools.
  - Key is to understand concept of protocol layering.
- OSI stack model vs. Middleware.
  - Classic network programming
  - Remote procedure call

A UNIX machine for the Course

- 162.105.146.57
  - dsYOURID/******
- If you do not use a unix-based OS, you may need
  - http://sewm.pku.edu.cn/tool/forWin/putty/putty.exe
  - http://sewm.pku.edu.cn/tool/forWin/winscp400.exe

Textbooks

  - book1, [Coulouri, 2005]
  - book2, [Birman, 2005]
  - book3, [Tanenbaum, 2006]
  - book4, [Tanenbaum, 2002]
  - ......

References

- Chapter 1 & 2 of [Birman, 2005]
- Chapter 1 & 2 of [Tanenbaum, 2002]
- Intro to SUN-RPC of CS244B: Distributed Systems@Stanford
  - [http://www.scs.stanford.edu/07wi-cs244b/notes/l1.pdf](http://www.scs.stanford.edu/07wi-cs244b/notes/l1.pdf)
- Introduction to the design of a distributed system
- Remote Procedure Calls
  - [http://www.linuxjournal.com/article/2204](http://www.linuxjournal.com/article/2204)