Chapter 3: Processes: Outline

- Process Concept: views of a process
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Inter Process Communication (IPC)
  - Local
    - Pipe
    - Shared Memory
    - Messages (Queues)
  - Remote
    - Lower Level: Sockets, MPI, Myrinet
    - Higher Level: RPC, RMI, WebServices, CORBA,

Client-Server Remote Machine Communication Mechanisms

- Socket communication (Possible bonus project)
- Remote Procedure Calls (Project next week).
- Remote Method Invocation (Briefly, on your own)

Remote Procedure Calls (RPC)

- Inter-machine process to process communication
  - Abstract procedure calls over a network:
    » Abstract procedure calls over a network:
    - Abstract procedure calls over a network:
    - daemons at ports
    » Registered library calls (port mapper)
    » Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (port mapper)
    - Registered library calls (port mapper)
  - Remote
    - Registered library calls (portmapper)
Remote Procedure Calls (RPC)

- **RPC High level view:**
  - Calling process attempt to call a 'remote' routine on server
  - Calling process (client) is suspended
  - Parameters are passed across network to a process server
  - Server executes procedure
  - Return results across network
  - Calling process resumes

- Usually built on top sockets (IPC)
- *stubs* – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.

Client/Server Model Using RPC

- Each RPC invocation by a client process calls a **client stub**, which builds a message and sends it to a **server stub**.

RPC Association Between Machines

- Association between remote and local host
  - 5 tuple
    - *(protocol, local-address, local-process, foreign-address, foreign-process)*
    - **Protocol**: transport protocol typically TCP or UDP, needs to be common between hosts
    - **Local/foreign address**: Typically the IP address
    - **Local/foreign process**: Typically the port number (not PID)
### Binding

- RPC application is packed into a program and is assigned an identifier (Port)
- Portmap: allocate port numbers for RPC programs

### Execution of RPC

- Client Process
- Portmapper
- Server Process

### Remote Procedure Calls

- Machine independent representation of data:
  - Differ if most/least significant byte is in the high memory address
  - External data representation (XDR)
    - Allows more complex representation that goes beyond: `htonl()` routines.
- Fixed or dynamic address binding
  - Dynamic: Matchmaker daemon at a fixed address (given name of RPC returns port of requested daemon)

### Tutorial (linux journal)

- `rpcgen` generates C code from a file written in 'RPC language' `<name>.x`, e.g., `avg.x`

<table>
<thead>
<tr>
<th>Default output rpcgen</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header file</td>
<td><code>&lt;name&gt;.h</code></td>
<td>avg.h</td>
</tr>
<tr>
<td>XDR data type translate routines (from type in <code>.h file</code>)</td>
<td><code>&lt;name&gt;_xdr.c</code></td>
<td>avg._xdr.c</td>
</tr>
<tr>
<td>stub program for server</td>
<td><code>&lt;name&gt;_svc.c</code></td>
<td>avg_svc.c</td>
</tr>
<tr>
<td>stub program for client</td>
<td><code>&lt;name&gt;_clnt.c</code></td>
<td>avg_clnt.c</td>
</tr>
</tbody>
</table>

- Application programmer (you) write code for:
  - Client routine (main program)
    - `ravg <host> <parameters>`
  - Server program (e.g., actual code to compute average)
    - `avg_proc.c`
Application Routines of Interest

- **Server Routine:**
  - `average_1_svc(input_data, )`:
    - A `avg_proc.c` routine that is called from the server stub that was generated by `rpcgen`

- **Client Routine:**
  - `average_prog_1()`:
    - Local routine that parses parameters and that ultimately calls a 'local' `average_1` routine from generated code in `avg_clnt.c` that packs parameters (also uses routines in `avg_xdr.c` and sends code to server).

### avg.x: RPC language file

```c
const MAXAVGSIZE = 200;
struct input_data
{
    double input_data<200>;
};
typedef struct input_data input_data;

program AVERAGEPROG {
    version AVERAGEVERS {
        double AVERAGE(input_data) = 1;
    } = 1;
} = 22855; /* port number */
```

### ravg.c: Client Program (1)

```c
/* client code - calls client stub, xdr client, xdr server, server stub, server routine */
#include "avg.h" /* header file generated by rpcgen */
#include <stdlib.h>

/* local routine client prototype can be whatever you want */
void averageprog_1( char* host, int argc, char *argv[] )
{
    CLIENT *clnt; /* client handle, rpc.h */
    double f, kkkkkk *result_1, *dp,
    char *endptr;
    int i;
    input_data average_1_arg; /* input_data rpc struct */
    average_1_arg.input_data.input_data_val = (double*) malloc(MAXAVGSIZE* sizeof(double));
    dp = average_1_arg.input_data.input_data_val; /* ptr to beginning of data */
    average_1_arg.input_data.input_data_len = argc - 2; /* set number of items */
    for( i = 1; i <= argc - 2; i++ )
    {
        /* str to d ASCII string to floating point number */
        f = strtod( argv[i+1], &endptr);
        printf( "value = %e\n", f);
        *dp = f;
        dp++;
    }
```

### ravg.c: Client Program (2)

```c
/* client code - calls client stub, xdr client, xdr server, server stub, server routine */
#include "avg.h" /* header file generated by rpcgen */
#include <stdlib.h>

/* local routine client prototype can be whatever you want */
void averageprog_1( char* host, int argc, char *argv[] )
{
    CLIENT *clnt; /* client handle, rpc.h */
    double f, kkkkkk *result_1, *dp,
    char *endptr;
    int i;
    input_data average_1_arg; /* input_data rpc struct */
    average_1_arg.input_data.input_data_val = (double*) malloc(MAXAVGSIZE* sizeof(double));
    dp = average_1_arg.input_data.input_data_val; /* ptr to beginning of data */
    average_1_arg.input_data.input_data_len = argc - 2; /* set number of items */
    for( i = 1; i <= argc - 2; i++ )
    {
        /* str to d ASCII string to floating point number */
        f = strtod( argv[i+1], &endptr);
        printf( "value = %e\n", f);
        *dp = f;
        dp++;
    }
```

```c
    if (clnt == NULL)
    {
        clnt_pcreateerror( host ); /* rpc error library */
        exit(1);
    }
    /* now call average routine 'just' like a local routine, but this will now go over network */
    *average_1 is defined in the client stub in avg_clnt.c that was generated by rpcgen
    * send in ptr to the parameters or args in first field, and client handle in second
    * field (created in clnt_create) average_1 ultimately calls clnt_call() macro see
    * man rpc, then calls the remote routine associated with the client handle
    * so AVERAGEPROG, VERSION */
    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 );
}
```

```c
/* next slide main() */
```
```c
int main( int argc, char* argv[] )
{
    char *host;
    /* check correct syntax */
    if( argc < 3 )
    {
        printf( "usage: %s server_host value ...
", argv[0]);
        exit(1);
    }
    if( argc > MAXAVGSIZE + 2 )
    {
        printf("Two many input values
" );
        exit(2);
    }
    /* host name is in first parameter (after program name) */
    host = argv[1];
    averageprog_1( host, argc, argv);
}
```

```c
#include <rpc/rpc.h>
#include "avg.h"
/* avg.h generated rpcgen */
#include <stdio.h>
/* run locally on 'server' called by a remote client. */
static double sum_avg;
/* routine notice the _1 the version number and notice the client handle, not used here, but * still needs to be a parameter */
double * average_1( input_data *input, CLIENT *client)
{
    /* input is parameters were marshaled by generated routine */
    /* a pointer to a double, set to beginning of data array */
    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++ ) /* iterate over input */
    {
        sum_avg = sum_avg + *dp; /* add what ptrs points to ('*' gets content ) */
        dp++;
    }
    sum_avg = sum_avg / input->input_data.input_data_len;
    return( &sum_avg );
} /* end average_1 */ /* next is routine called from server stub generated by rpcgen */
```

```c
#include <rpc/rpc.h>
#include "avg.h" /* avg.h generated rpcgen */
#include <stdio.h>
/* run locally on 'server' called by a remote client. */
static double sum_avg;
/* routine notice the _1 the version number and notice the client handle, not used here, but * still needs to be a parameter */
double * average_1( input_data *input, CLIENT *client)
{
    /* input is parameters were marshaled by generated routine */
    /* a pointer to a double, set to beginning of data array */
    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++ ) /* iterate over input */
    {
        sum_avg = sum_avg + *dp; /* add what ptrs points to ('*' gets content ) */
        dp++;
    }
    sum_avg = sum_avg / input->input_data.input_data_len;
    return( &sum_avg );
} /* end average_1 */ /* next is routine called from server stub generated by rpcgen */
```
### Compilation on client

```
rpcgen avg.x # generates:
   # avg_clnt.c, avg_svc.c, avg_xdr.c, avg.h
 gcc ravg.c -c       # -c generates .o files
 gcc avg_clnt.c -c
 gcc avg_xdr.c -c
 gcc -c ravg ravg.o avg_clnt.o avg_xdr.o -lns1
```

### Compilation on server

```
rpcgen avg.x # generates:
   # avg_clnt.c, avg_svc.c, avg_xdr.c, avg.h
 gcc avg_proc.c -c
 gcc avg_svc.c -c
 gcc -o avg_svc avg_proc.o avg_svc.o avg_xdr.o -lns1
```

### .rhost

- Directly under your home directory on each machine (client and server) create a file named: `.rhost`
- Add two or more lines in the format:
  `<machine_name>   <loginname>`
- For example I added 3 lines:
  ```
  odin maria
  herc maria
  atlas maria
  ```

### Running

```
{maria:herc} avg_svc
```

```
{maria:odin} ravg atlas.cs.uga.edu 1 2 3 4 5
```
Resources

1. [http://www.cs.cf.ac.uk/Dave/C/node34.html](http://www.cs.cf.ac.uk/Dave/C/node34.html) (RPCgen)
2. [http://www.cs.cf.ac.uk/Dave/C/node27.html](http://www.cs.cf.ac.uk/Dave/C/node27.html) (Sh. Mem)
3. [http://www.linuxjournal.com/article/2204?page=0,2](http://www.linuxjournal.com/article/2204?page=0,2)

(1) Nice tutorial on RPC
(2) (2) Shared Memorh
(3) Linux journal tutorial uses avg.x
(4) Beej’s Guide

Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.
- Possible to Pass Objects (remote, local) as parameters to remote methods (via serialization).

Marshalling Parameters

- Client invoke method: someMethod on a remote object Server

```java
val = server.someMethod(A, B)

boolean someMethod (Object x, Object y) {
    implementation of someMethod
    ...
}
```

Diagram:

- JVM
- Java program
- remote method invocation
- remote object