CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2003

Lecture 21: Remote Procedure Call (RPC)

21.0 Main Point

- Send/receive
- One vs. two-way communication
- Remote Procedure Call
- Cross-address space vs. cross-machine communication

21.1 Send/Receive

How do you program a distributed application? Need to synchronize multiple threads, only now running on different machines (no shared memory, can't use test&set).

Key abstraction: send/receive messages. (Note: this abstraction is similar to "read" and "write" of the file I/O interface.)

Atomic operations: send/receive – doesn't require shared memory for synchronizing cooperating threads.

Questions to be answered:

- Blocking vs. Non-blocking calls
- Buffered vs. Non-buffered I/O
- How does addressing work?

Mailbox – temporary holding area for messages (ports)

Send(message, mbox) – put message on network, with name of mbox for where it is to go on far end

When can Send return?

1. When receiver gets message? (i.e., acknowledgement received)

2. When message is safely buffered on destination node?
3. Right away, if message is buffered on source node?

There are really two questions here:

1. When can the sender be sure that the receiver actually received the message?
2. When can the sender re-use the memory containing the message?

Receive(buffer, mbox)

- Wait until mbox has message in it, then copy message into buffer, and return.
- When packet arrives, OS puts message into mbox, wakes up one of the waiters.

Note that send and receive are atomic (the OS ensures this abstraction).

- Never get portion of a message (all or nothing)
- Two receivers can't get same message

21.2 Message styles

Using send/receive for synchronization:

```
Producer:
     int msg1[1000]

     while (1)
          prepare message  // make coke
          send(msg1, mbox)

Consumer:
     int msg[1000]

     while (1)
          receive(msg2, mbox)
          process message  // drink coke
```

No need for producer/consumer to keep track of space in mailbox – handled by send/receive.

What about two-way communication? Request/response. For instance, "read a file" stored on a remote machine or request a web page from a remote web server.

Also called: client-server. Client = requester; server = responder. Server provides "service" (file storage) to the client

Request/response:

```
Client: (requesting the file)
     char response[1000];

     send("read rutabaga", mbox1);
     receive(response, mbox2);

Server:
     char command[1000], answer[1000];

     receive(command, mbox1);
     decode command
     read file into answer
     send(answer, mbox2);
```

Server has to decode command, just as OS has to decode message to find mbox.

What if file is too big for response (e.g., maximum message size is 1KB)?

Use "remote file access" protocol:

```
send("open rutabaga", mbox1);
receive(status, mbox2);
if (!streq(status, "OK")) Error(status);
while (TRUE) {
     send("read 1024", mbox1);
     receive(buf, mbox2);
     if (!streq(status, "OK")) {
          if (streq(status, "EOF")) break;
          Error(status);
     }
     ...
}
send("close", mbox1);
receive(status, mbox2);
```

Problem: I/O is not the most natural programming model for many situations. What is?

21.3 Remote Procedure Call (RPC)

Call a procedure on a remote machine.

Client calls:

```
remoteFileSys->Read("rutabaga")
```

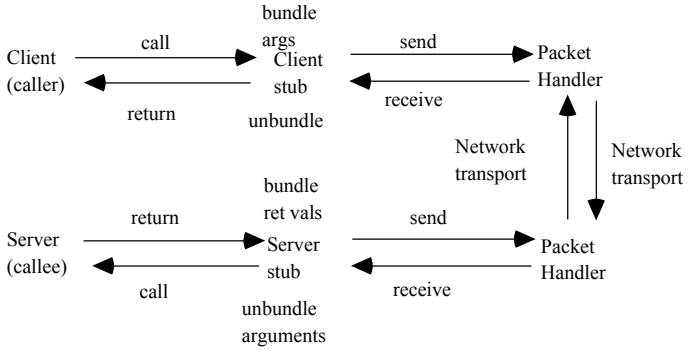
Translated into call on server:

```
fileSys->Read("rutabaga")
```

Implementation:

- Request-response message passing
- "Stub" provides glue on client/server.
    - Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values

- Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- Marshalling involves (depending on the system): converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.




```
Client stub:
     Build message
     Send message
     Wait for response
     Unpack reply
     Return result

Server stub:
     Create N threads to wait for work to do
     Loop:
          Wait for command
          Decode and unpack request parameters
          Call procedure
          Build reply message with results
          Send reply
```


21.3.1 Comparison between RPC and procedure call

What's equivalent?

Parameters – request message
Result – reply message
Name of procedure – passed in request message
Return address – mbox2

21.3.2 Implementation issues

Stub generator – generates stubs automatically. For this, only need procedure signature: types of arguments, return value. Generates code on client to pack message, send it off; on server to unpack message, call procedure.

Input: *interface definitions* in an interface definition language (IDL).

Output: stub code in the appropriate source language (C, C++, Java, ...)

What if client/server machines are different architectures? Or, they use different programming languages?

Examples of modern RPC systems:

- CORBA (Common Object Request Broker Architecture)
- DCOM (Distributed COM)
- RMI (java Remote Method Invocation)

How does client know which mbox to send to? Binding

Static – fixed at compile time (C)
Dynamic – fixed at runtime (Lisp, RPC)

In most RPC systems, dynamic binding via name service. Name service provides dynamic translation of service -> mbox

Why runtime binding?

- Access control – check who is permitted to access service
- Fail-over – if server fails, use another

What if there are multiple servers – can they use same mbox? OK, if no state carried forward from one call to the next. For example, open, seek, read, close – each uses context of previous operation.

What if multiple clients?

All would receive results in same mbox

Fix by passing pointer to mbox in request message

(this is like pushing the return PC on the stack)

21.4 Problems with RPC

RPC provides location transparency, except:

21.4.1 Failures

Different failure modes in distributed system than on single machine.

Several kinds of failures:

– User-level bug causes address space to crash

– Machine failure, kernel bug causes all processes on same machine to fail

– Earthquake causes all machines to fail

Before: whole system would crash. Now: one machine can crash, while others stay up. If file server goes down, what do the other machines do?

Can easily result in inconsistent views of the world:

- Did my cached data get written back or not?
- Did you do what I requested or do I need to re-request it later?

The answer: distributed transactions; if only they were cheaper! (recall: cost of two-phase commit) In many cases, can deal with "at most once" semantics and fail fast behavior.

21.4.2 Performance

Cost of a procedure call << same-machine RPC << network RPC

Means programmers must be aware that RPC is cheap, but it's not free.

Caching can help, but that can make failure handling more complex.

In a distributed system, you have to face up to these two problems. Illustrate in lecture on network file systems.

On same machine (for example, microkernel OS), the tradeoff is – what do you gain for the loss of performance?

21.5 Cross-Domain Communication and Location Transparency

How do address spaces communicate with one another?

- Semaphores, etc.
- File system
- Shared memory
- Pipes (1-way communication)
- "Remote" procedure call (2-way communication)

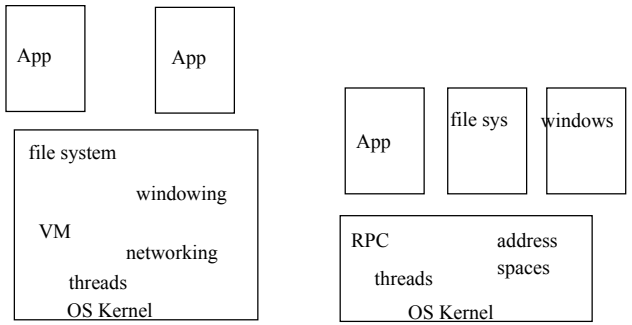
RPC's can be used to communicate between address spaces on different machines or between address spaces on the same machine.

This provides location transparency:

- Services can be run wherever it's most appropriate.
- Access to local and remote services looks the same.

21.5.1 Microkernel operating systems

Example: split kernel up into application-level servers. File system looks like it is remote, even though on the same machine



Monolithic Structure

Microkernel Structure

Why split the OS into separate domains?

- Fault isolation – bugs are more isolated (build a firewall)
- Enforces modularity – allows incremental upgrades of pieces of software (client or server)
- Location transparent – service can be local or remote

  For example in the X windowing system:

  Each X client can be on a separate machine from X server. Neither has to run on the machine with the frame buffer. This is how X terminals work – essentially, CRT screens with multiple windows.

21.5.2 OLE (Object Linking and Embedding)

Another example: Desktop publishing.

Need editor, spreadsheet, document formatter, database, etc.

Now bundled, but might be composed of separable pieces.

Principle behind Microsoft OLE – allow you to mix and match applications, which cooperate via defined RPC interface.