Process Communication Paradigms

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE855
Communicating to remote processes
- message passing
  - simplest form of communication
  - client/server model
  - messages explicitly manipulated by the user
- remote procedure call (RPC)
  - procedure calls + stubs
  - implicit bi-directional flow of information
  - messages implicit
- transactions
  - synchronization and serialization of communication
  - messages implicit, handle multiple messages (chapter 3)

Client-Server Model
- Server: provides some service to client processes
  - a process that “listens” to a port
  - accepts connections from a client
  - is passive -- waits for a request
- Client: requests services
  - must know name of the service (an address)
  - establishes connection with server
  - requests services (provide data, perform calculations, etc.)
Client-Server Model (cont.)

- Addressing a server
  » a "well-known socket" address
  » port address hardwired
    ♦ port is just an address
    ♦ operating system gets a message with a port address
    ♦ sends message to the code handling the port
  » bind address to a name
  » client requests the service by name or address

Client-Server Model (cont.)

- Limited number of servers
  » clients can be from anywhere
  » knowledge of “well-known address” is all that’s needed

- Servers vs. services
  » service: a software facility (sometimes implemented as
    a set of servers)
  » server: software running on one machine

- Some problems with C/S model:
  » extendibility: as nodes added to system, servers may
    become over-loaded with clients
  » single point of failure
  » multiple servers increase costs
Dimensions of the Client/Server Model

- Addressing
  - how to find server addresses
- Blocking and Non-Blocking methods
  - can either block or continue when sending or receiving messages
- Buffered vs. Unbuffered methods
  - choice of buffering messages at the server or with the kernel
- Reliable vs. Unreliable methods
  - choices on when to acknowledge a message

C/S Addressing Schemes

- Process-to-process message passing
  - machine.process hardwired into client
    - i.e. the "well-known address"
  - Unix sockets
    - not transparent to user (programmer)
    - if addressed server machine is down, system breaks
Broadcasting for process addressing
- Each process has a unique global address
  - Central server assigns a number to the process
  - OR a process chooses from a sparse address space
- A client broadcasts a 'locate packet' with the server address it needs
  - The server responds with 'here I am'
  - The client caches the address
  - Sends subsequent messages to that address
- Problem: broadcast takes up bandwidth

Name service approach
- A name server holds name-address mappings
  - A server must register this mapping
  - A client does a look-up on the name service
    - Caches address (machine address)
  - Uses address from then on
  - Problem: name service is a central component
## Blocking vs. Non-Blocking Primitives

- **Blocking (synchronous) primitives**
  - when message is sent, sending process waits until message has been successfully sent
  - receiving message is blocked until message is in process’ buffer
  - clearest semantics, easiest to implement

- **Non-Blocking (asynchronous) primitives**
  - `send` returns immediately
  - can’t use buffer until message sent
  - sender doesn’t know when that’s happened
  - copy into kernel buffer, then re-use the process buffer
  - but overhead of copy is prohibitive
  - `receive` gets the buffer and returns control
  - process must determine if buffer has been written to
  - can use an explicit `wait` to block or `test` to poll kernel

## Buffered vs. Unbuffered Primitives

- **Buffered (buffer at kernel)**
  - server requests a mailbox from the kernel
  - `receive` removes a message from the mailbox
  - mailboxes can fill up
  - messages can be discarded or kept for a time

- **Unbuffered (buffer at server)**
  - kernel copies message to process buffer and unblocks process
  - What if message is received before process does a `receive`?
    - discard the message (sender will re-transmit)
    - keep message for a time period
  - What happens while the server is processing a previous request?
    - multiprogrammed servers
Reliable vs. Unreliable Primitives

- **Reliable**
  - acknowledge each message
  - results in 4 messages per c/s exchange
  - reply serves as implicit acknowledge
  - reply from server is acknowledgment for the client
  - can choose to ask the server’s reply
    - if not, server sends reply again
    - 2-3 messages per c/s exchange
  - but hard for client to distinguish between a slow server and one that’s down
  - server sends ack only if service takes too long
  - after a time-out, server sends explicit ack

- **Unreliable**
  - no acknowledgments
  - reliability up to users (program designers)
    - note any reliability must be distributed

Client/Server Design Issues

- Many trade-offs between choices
  - acknowledge only entire messages
    - fewer ack messages
    - but recovery is more complicated or inefficient
    - works bet for reliable networks
  - acknowledge individual packets
    - more ack messages
    - but recovery is easier, more efficient (less packets re-transmitted)
    - may want to use on unreliable networks

- Packet exchanges
  - packet types for “I am alive,” “Try again,” etc.
  - use to design different protocols
Remote Procedure Calls

- Procedure calls for remote communication
  - call: blocking send
  - called procedure: blocking read, return results
  - allows type-checked communication
    - compiler detects inconsistencies
    - treated like any other procedure call

- Language-level calls on each end of communication
  - caller:
    remote procedure X (parameters)
  - callee:
    int remote procedure X (parameters)

Remote Procedure Calls
Procedure-to-message conversion

- Server stub (caller)
  - packs parameter into frame
  - receives reply, unpacks

- Client stub (callee)
  - unpacks parameters
  - sends reply

<table>
<thead>
<tr>
<th></th>
<th>return</th>
<th>Client stub</th>
<th>local call</th>
<th>transmit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>receive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unpack</td>
<td>Client stub</td>
<td>pack params</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>transmit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Server stub</td>
<td>pack reply</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RPC Stubs

◆ Stub compiler
  » programmer specifies server specification
  » calls compiler with RPC switches
    ◦ cc prog.c -lrpcsvc -lsun
  » compiler automatically creates stub

◆ Server stub
  » also created with stub compiler
  » server needs to register its services

◆ Server specification
  » can choose from a set of parameter types
  » or can create own

---

Low-Level RPC - Server

```c
#include <stdio.h>
#include <rpc/rpc.h>

int *nuser (int *indata) {
    int total;
    total = *indata + 2;
    printf("Input data was: %d\n", *indata);
    return (&total);
}

main() {
    registerrpc (200012, 2, 2, nuser, xdr_int, xdr_int);
    svc_run();
    printf("ERROR, svc_run() returned!!\n");
    exit(1);
}
```
Low-Level RPC - Client

```c
#include <stdio.h>
#include <rpc/rpc.h>

main (int argc, char *argv[]) {
    int outdata;
    int stat, indata;
    if (argc < 2) {
        printf ("Usage: rpc-c host\n");
        exit(0);
    } indata = 2;
    if (stat = callrpc (argv[1], 200012, 2, 2, xdr_int, &indata, xdr_int, &outdata) != 0) {      clnt_perrno(stat);
        exit(1);
    } printf ("result received: %d\n", outdata);
    exit(0);
}
```

Higher-Level RPC - rpcgen

```c
/* msg.x: used by rpcgen()
to generate;
msg_svc.c (server stub) and msg_clnt.c (client stub)
these were compiled with the client and server code
gcc server.c msg_svc.c -lsun -o serv
gcc client.c msg_clnt.c -lsun -o clnt
serv was run in the background (serv &)
clnt was given a host name and a number (i.e. clnt cse 4)
*/

program MESSAGEPRG {
    version MESSAGEVERS {
        string PRINTMESSAGE(string) = 1;
        PRINTMESSAGE(string) = 0x2000099;
    } = 1;
}
```
#include<stdio.h>
#include<rpc/rpc.h>
#include "msg.h"

/* SERVER */
/* msg_proc.c*/

char ** printmessage_1(char **msg) {
static char *result;
int rec_num;
char buffer[80];
rec_num = atoi(*msg);
for (i=0;i<rec_num;i++) {
    strcpy(&buffer[i*6],"HELLO ");
}

printf("SERVER RECEIVED %d, SENT: %s 
", rec_num,buffer);
result = buffer;
return (&result);
}

Higher-Level RPC - Client

#include<stdio.h>
#include<rpc/rpc.h>
#include "msg.h"

main(int argc, char **argv[]) {
    CLIENT *cl; char **result; char *server; 
    char *message; server = argv[1]; /* Should check for 2 arguments!!! */ 
    message = argv[2];
    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
    if (cl == NULL) {
        clnt_pcreateerror(server);
        exit(1);
    }
    result = printmessage_1(&message, cl); 
    if (result == NULL) {
        clnt_perror(cl,server);
        exit(1);
    }
    if (*result == 0) {
        printf("message unavailable 
");
        exit(1);
    }
    printf("CLIENT RECEIVED MESSAGE: %s\n",*result);
    exit(0);
}
### RPC Parameter Passing

- **Call-by-value**
  - easy, just pass the value

- **Call-by-reference**
  - address, not value is passed
  - address must make sense on remote machine

- **Call-by-copy/restore**
  - copy address space in message
    - for example, must pass entire array, not just pointer to array
  - server manipulates data in its address space
  - client must overwrite the data structure when reply is received
  - What about a linked list?

### Parameter Passing (cont.)

- **Call-by-reference through messages**
  - whenever pointer referenced, send message to client to get value
  - client stub must be set up to answer server

- **Parameter marshaling**
  - problem: different machines represent numbers, characters, etc., differently
  - network-wide canonical form
    - each machine only responsible for converting from canonical to local form, but may end up doing unnecessary conversions
    - from ‘big endian’ to canonical to ‘big endian’...
  - client identifies message format
    - only server needs conversion routines
Client/Server Binding

- Duration of connection
  - make connection each time service needed
  - keep virtual circuit active between calls
- Communication binding
  - static: direct communication determined at compile time
  - dynamic: communicate through a name service
    - server exports (registers) its services with the binder
    - a kind of name server
    - client makes an import call to binder
    - if server exists, binder gives address to client

Handling RPC Failures

- When RPC fails:
  - transparency is lost
  - client programmers may want to make exception handlers (transparency is lost)
- Lost request messages
  - kernel re-sends message after time out
- Reply message is lost
  - idempotent operations - just ask for service again
  - non-idempotent operations are more difficult
    - assign request number to each request
    - server refuses to re-do requests
Handling RPC Failures (cont.)

- Client can’t locate server
  » need exception handlers
- Server crashes
  » server crashes (some time) after receiving request
  » at least once semantics
  » at most once semantics
  ◆ difficult to guarantee
- Client crashes
  » client crashes before server replies
  » server is active - but can’t send result
  ◆ known as an orphan
  » various methods to remove the orphans

RPC Implementation Issues

- Protocols
  » same issues as in client-server
  » connectionless protocols dominate
  ◆ performance is needed, LANS are reliable
  » customized RPC protocols are common
- Acknowledgments
  » stop-and-wait
  » blast
  ◆ client sends all packets
  ◆ server acknowledges with one ack
  ◆ re-send entire message vs. selective repeat
  ◆ network chips don’t always have capacity for blast
Critical path analysis
- context switching most expensive
- busy wait, then multiprogramming suffers
- also copying between user and kernel address spaces

Copying between address spaces
- varies from 1 to 8 copies per message
- changing memory map to achieve “copying”
  - kernel changes memory map so buffer is now in user’s memory map
  - user program has access to memory without copying
  - message needs to be on page boundaries

Timer management
- lot of time-outs - very CPU intensive
  - fortunately few need exact time
  - sorted list
  - expensive to update when reply received
  - sweep algorithms
    - each process in process table has ‘timer’ field
    - zero means timer is off
    - kernel scans process table for expired timers
**Problems Inherent to RPC**

- **Global variables**
  - remote procedures don’t have access to globals
  - Will RPC ever achieve full transparency?

- **Pipe structures**
  - `p1 < f1 | p2 | p3 > f2`
  - read-driven: each program is an active client requesting a read
  - `p1` requests read from `f1`
  - `p2` requests read from `p1`
  - `p3` requests read from `p2`
  - file server needs to act as a client requesting read from `p3` - but it’s role is as a server!
  - write-driven: mirror image problem

---

**RPC vs. Message Passing**

- **Message Passing**
  - user is explicitly concerned with message manipulation
  - need to define syntax and semantics
  - flexibility, any semantics can be defined

- **RPC**
  - message passing accomplished transparently
  - syntax and semantics are given
  - semantics are set
  - blocking send by client
  - blocking read by server

- **Performance issues still an open issue**
  - concurrency not supported well by RPC
  - applications may experience different performance differences
  - implementation is crucial and not yet well-established
**Group Communication**

- **RPC:** communication only involves two processes  
  » note: not so for general client/server model
- **Group definition**  
  » set of processes working together  
  » processes free to join or leave group  
  » messages sent to all in a group  
  » only group has access to communication

---

**Group Communication (cont.)**

- **Multicasting**  
  » special network address used to define groups  
  » machine listens only if part of group
- **Broadcasting**  
  » message sent to all machines  
  » kernel determines if any processes in the group
- **Unicasting**  
  » send point-to-point message to all in group
**Group Communication Design Issues**

- Closed vs. open groups
  - closed doesn’t allow outside messages - parallel processing

- Peer vs. hierarchical groups
  - decision making in groups
  - all participate vs. coordinator
  - with coordinator, must have election algorithm if coordinator dies

- Group membership
  - group server to maintain group status
    - falls into centralized trap
  - member crashes must be discovered
  - when joining group, must get all messages immediately
    - when leaving, cannot receive any more messages

**Group Communication Design Issues (cont.)**

- Group addressing
  - multicast
    - message sent to all machines with process in group
  - broadcast
    - message must be discarded by machines not in group
  - unicast
    - kernel sends message to each machine
  - group members send to group members
    - requires each member to maintain group list
Send and receive primitives
» problem: there are potentially $n$ different replies
» solution: treat replies as separate messages
   ♦ but difficult to merge with RPC
   ♦ different calls for group communication
   ♦ group_send
   ♦ group_receive

Atomic broadcast
» either all get the message or none do
   ♦ simple semantics: if one member doesn’t get message, just re-send
   ♦ no need for selective re-send
   » difficult to achieve in practice
   » one method:
      ♦ sender sends message to all in group
      ♦ if receiver has seen message before, discard it
      ♦ if message is new to receiver, send to all in group
      ♦ all (non-crashed) processes will get message
      ♦ lots of overhead in the form of unnecessary messages
Group Communication
Design Issues (cont.)

◆ Message ordering
  » arrival times over LAN is nondeterministic
  » global time ordering
  » consistent time ordering

◆ Overlapping groups
  » global time ordering only within a group

◆ Scalability
  » sending multicasts & broadcasts to interconnected LANs
    ♦ gateways just forward the message
    ♦ messages will be repeated
  » packets can be actively transmitting simultaneously
    when LAN interconnected
    ♦ destroys global ordering

Group Communication
Design Issues (cont.)