CprE 450/550x
Distributed Systems and Middleware

# Inter-process Communication

Yong Guan
3216 Coover
Tel: (515) 294-8378
Email: guan@ee.iastate.edu
February 3, 2004



## Readings for Today's Lecture

- References
  - **Chapter 2 of "Distributed Systems: Principles and Paradigms"**
  - **Chapter 4 of "*Distributed Systems: Concepts and Design*"**
  - **Chapter 14 & Chapter 15 of "Advanced Programming in the UNIX Environment"**



## Interprocess Communication

- Primitives
- Message Passing: issues
- Communication Schemes



## Interprocess Communication (IPC)

lack of shared memory → communicate by sending messages

**Primitives for interprocess communication**

- message passing
  - **the RISC among the IPC primitives**
- remote procedure call (RPC)
  - **process interaction at language level**
  - **type checking**
- transactions
  - **support for operations and their synchronization on shared objects**

5

## Message Passing

- The primitives:

```
send expression_list to destination_identifier;

receive variable_list from source_identifier;
```

- Variations:

```
guarded receive:
    receive variable_list from source_id when B;

selective receive:
    select
      receive var_list from source_id1;
     |receive var_list from source_id2;
     |receive var_list from source_id3;
    end
```

6

## Semantics of Message-Passing Primitives

- blocking *vs.* non-blocking
- buffered *vs.* unbuffered
- reliable *vs.* unreliable
- fixed-size *vs.* variable-size messages
- direct *vs.* indirect communication

7

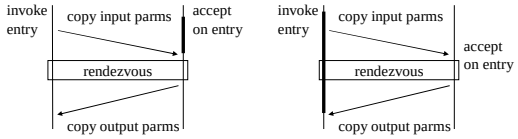
## Blocking *vs.* Non-Blocking Primitives

| | blocking | non-blocking |
|---|---|---|
| send | Returns control to user only after message has been sent, or until acknowledgment has been received. | Returns control as soon as message queued or copied. |
| receive | Returns only after message has been received. | Signals willingness to receive message. Buffer is ready. |
| *problems* | •*Reduces concurrency.* | •*Need buffering:*<br>•*still blocking*<br>•*deadlocks!*<br>•*Tricky to program.* |

8

## Buffered *vs.* Unbuffered Primitives

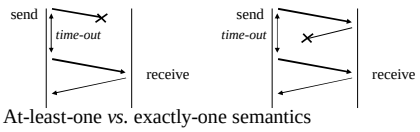
- Asynchronous send is never delayed
  **may get arbitrarily ahead of `receive`.**
- However: messages need to be buffered.
- If no buffering available, operations become blocking, and processes are synchronized on operations: **rendezvous**.



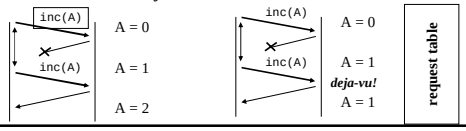
9

## Reliable *vs.* Unreliable Primitives

- Transmission problems:
  corruption loss duplication reordering
- Recovery mechanism: Where?
- Reliable transmission: acknowledgments




- At-least-one *vs.* exactly-one semantics




10

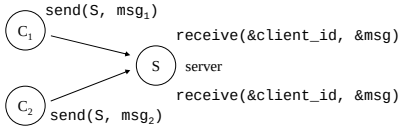
## Direct *vs.* Indirect Communication

- Direct communication:

```
send(P, message)
receive(Q, message)
```

- Variation thereof:

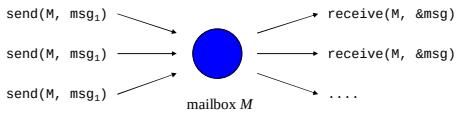
```
send(P, message)
receive(var, message)
```




11

## Direct *vs.* Indirect Communication (cont.)

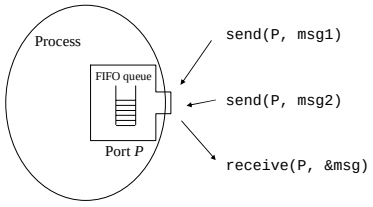
- Indirect communication:
  **Treat communication paths as first-class objects.**
- Mailboxes:




12

## Direct *vs.* Indirect Communication (cont.)

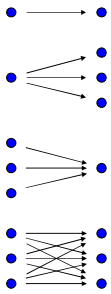
- Indirect communication (cont)
- Ports:
  **example: Accent (CMU)**




- multiple senders
- only one receiver
- access to port is passed between processes in form of capabilities

13

## Communication Schemes



one-to-one unicast

one-to-many multicast

many-to-one

many-to-many

14

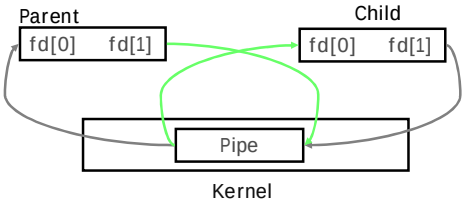
## Case Study: IPC on the Same Host

- Ways of Inter-process Communication
  - **Signal**
  - **Passing file descriptor between parent and child processes**
  - **UNIX IPC**
    - ✓ **Pipes**
    - ✓ **FIFOs**
    - ✓ **Stream Pipes**
    - ✓ **Named Stream Pipes**
    - ✓ **Message Queues**
    - ✓ **Semaphores**
    - ✓ **Shared Memory**

15

## Case Study: IPC on the Same Host (cont.)

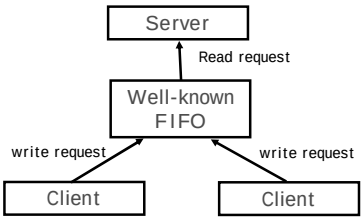
- Pipes
  - **Half-duplex**
  - **Only used between processes that have a common ancestor, e.g., parent and child processes.**




16

## Case Study: IPC on the Same Host (cont.)

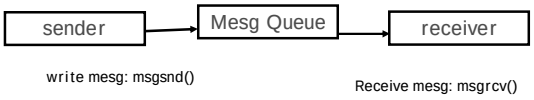
- FIFO (also called named pipe)
  - **Half-duplex**
  - **Can be used between unrelated processes (not necessarily between parent and child processes).**



17

## Case Study: IPC on the Same Host (cont.)

- Message Queues
  - **A linked list of messages stored in the kernel and identified by mesg queue id.**
  - **Not necessarily first-in first-out order**
  - **Can fetch messages based on type**
  - **Bi-directional**




write mesg: msgsnd()

Receive mesg: msgrcv()

18

## Case Study: IPC on the Same Host (cont.)

- Semaphore
  - **Not really a form of IPC as pipe, FIFOs, and message queues**
  - **A counter used to provide access to a shared data object for multiple processes**
    1. Test the semaphore that controls the resource
    2. If the value is positive, the process can use the resource and the value of semaphore decrements by one.
    3. If the value is 0, the process goes to sleep until the semaphore value is greater than 0.

19

## Case Study: IPC on the Same Host (cont.)

- Shared Memory
  - **Allow two or more processes to share a given region of memory.**
  - **Fastest IPC mechanism**
  - **Synchronization access**

20

## Case Study: IPC on the Same Host (cont.)

- Stream pipes
  - **Allow passing open file descriptors between processes (parent and a child)**
  - **Bi-directional**
- Similar to FIFO, we have named Stream Pipe

21

## Remote Procedure Call (RPC)

- Now we study RPC.

22

## Remote Procedure Call (RPC)

- Paradigms in building distributed applications
- The RPC model
- Primitives
- Issues
- Case study: Sun RPC

23

## Building Distributed Programs: Two Paradigms

Paradigms:

- Communication-Oriented Design
  - Start with communication protocol
  - Design message format and syntax
  - Design client and server components by specifying how they react to incoming messages

Problems:

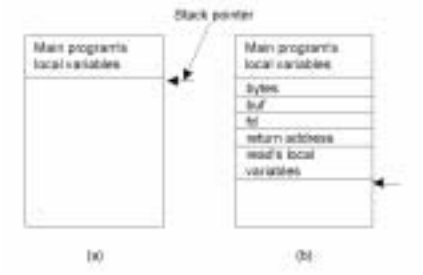
- Protocol-design problems
- Application components as finite-state machines !?
- Focus on communication instead of application!

- Application-Oriented Design
  - Start with application
  - Design, build, test conventional implementation
  - Partition program

- Concurrency

24

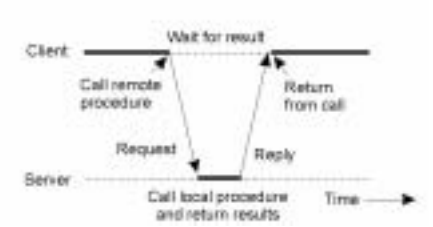
## Conventional Procedure Call




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

## RPC: Client and Server Stubs

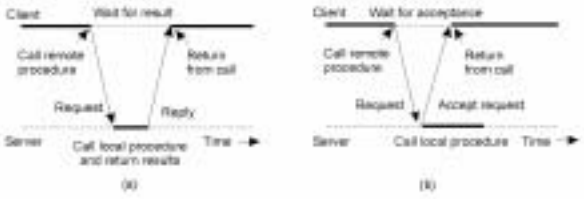





- Principle of RPC between a client and server program.

## Asynchronous RPC (1)

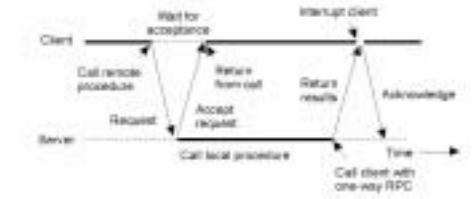





a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC

## Asynchronous RPC (2)



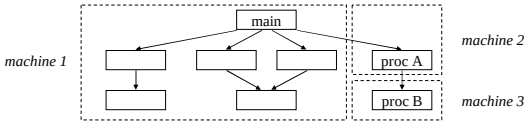



- A client and server interacting through two asynchronous RPCs

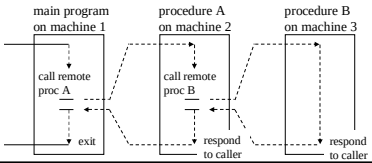
## Model of Execution for RPCs



- Procedure-call structure of a program




- Model of execution with remote procedure call



29

## RPC Properties

- Uniform call structure
- Type checking
- Full parameter functionality
- Distributed binding
- Recovery of orphan computations

30

## RPC Primitives

- Invocation at caller side

```
call service (value_args; result_args);
```

- Definition at server side
  - **declaration**

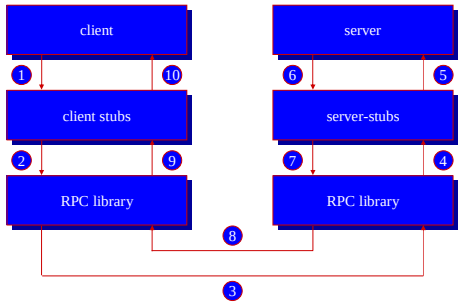
```
remote procedure service (in value_pars;
                          out result_pars);
begin body end;
```

  - **rendezvous statement**

```
accept service (in value_pars;
                out result_pars) -> body;
```

31

## Structure of an RPC Call




32

## 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

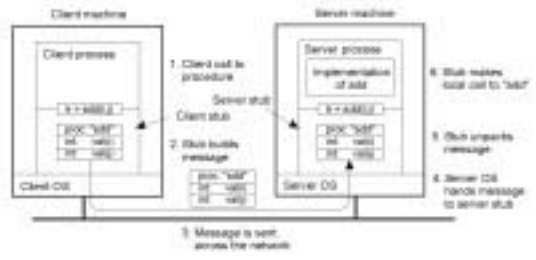
33

## RPCs: Issues

- Parameter passing
  - **value parameters**
  - **reference parameters?**
- Marshalling
  - **simple data types**
  - **complex data structures**
- Exception handling
  - **language dependent**
  - **need to deal with asynchronous events**

34

## Passing Value Parameters

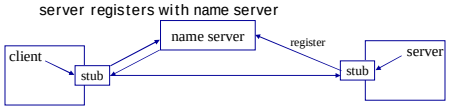


- Steps involved in doing remote computation through RPC

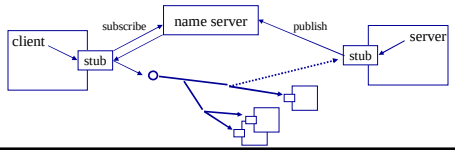
35

## Locating Servers

- Broadcast requests
  - broadcast call and process incoming replies
- Name servers
  - server registers with name server




- Combination: publish/subscribe

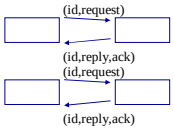



36

## Communication Protocols for RPC

- Reliable protocols: e.g. TCP
- Unreliable datagram protocols: e.g. UDP
- Specifically designed protocols: Example

Simple Call




Client times out and retransmits request. Three cases:

- request lost
- server still executing
- ack lost

Complicated Call

- long gaps between requests
  - acknowledge each message transmission separately

  or
  - periodically send "I-am-alive" message and use simple-call scheme.
- long messages (don't fit into packet)
  - segment message
  - segment-relative seq #'s
  - retransmission scheme for segments

37

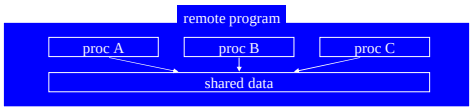
## RPC in Heterogeneous Environments

- Compile-time support
- Binding protocol
- Transport protocol
- Control protocol
- Data representation

38

## Case Study: SUN RPC

- Defines format for messages, arguments, and results.
- Uses UDP or TCP.
- Uses XDR (eXternal Data Representation) to represent procedure arguments and header data.
- Compiler system to automatically generate distributed programs.
- Remote execution environment: remote program.




Mutually exclusive execution of procedure in remote program.

39

## Identifying Remote Programs and Procedures

- Conceptually, each procedure on a computer is identified by pair :

  *(prog, proc)*

  ***prog*: 32-bit integer identifying remote program**

  ***proc*: integer identifying procedure**
- Set of program numbers partitioned into 8 sets.

| | |
|---|---|
| 0x00000000 - 0x1fffffff | assigned by SUN |
| 0x20000000 - 0x3fffffff | assigned by local system manager |
| 0x40000000 - 0x5fffffff | temporary |
| 0x60000000 - 0xffffffff | reserved |

- Multiple remote program versions can be identified:

  *(prog, version, proc)*

40

## Example RPC Program Numbers

| name | assigned no | description |
|---|---|---|
| portmap | 100000 | port mapper |
| rstatd | 100001 | rstat, rup, perfmeter |
| rusersd | 100002 | remote users |
| nfs | 100003 | network file system |
| ypserv | 100004 | yp (NIS) |
| mountd | 100005 | mount, showmount |
| dbxd | 100006 | DBXprog (debug) |
| ypbind | 100007 | NIS binder |
| walld | 100008 | rwall, shutdown |
| yppasswdd | 100009 | yppasswd |

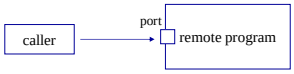
41

## Communication Semantics

- TCP or UDP ?
- Sun RPC semantics defined as function of underlying transport protocol.
  - **RPC on UDP: calls can be lost or duplicated.**
- *at-least-once* semantics if caller receives reply.
- *zero-or-more* semantics if caller does not receive reply.
- Programming with zero-or-more semantics: *idempotent* procedure calls.
- Sun RPC retransmission mechanism:
  - **non-adaptive timeouts**
  - **fixed number of retransmissions**

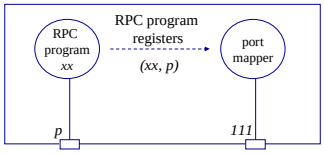
42

## Remote Programs and Protocol Ports




program_id (32 bit) *vs.* port_id (16 bit)

- Dynamic port mapping: *RPC port mapper*




port currently used by this RPC program — well-known port for port manager

43

## Sun RPC Message Format: XDR Specification

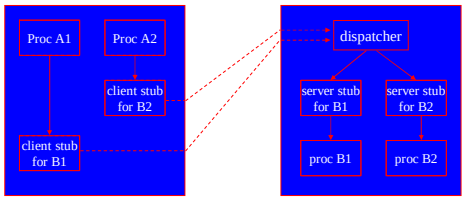
```
enum msg_type {  /* RPC message type constants */
      CALL  = 0;
      REPLY = 1;
};

struct rpc_msg {       /* format of a RPC message     */
  unsigned int mesgid; /* used to match reply to call */
  union switch (msg_type mesgt) {
      case CALL : call_body  cbody;
      case REPLY: reply_body rbody;
  } body;
};

  struct call_body {  /* format of RPC CALL              */
    u_int rpcvers;    /* which version of RPC?           */
    u_int rprog;      /* remote program number           */
    u_int rprogvers;  /* version number of remote prog */
    u_int rproc;      /* number of remote procedure    */
    opaque_auth cred; /* credentials for called auth.  */
    opaque_auth verf; /* authentication verifier       */
    /* ARGS */
  };
```

44

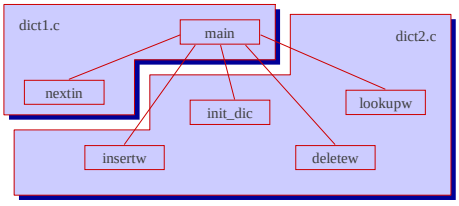
## Message Dispatch for Remote Programs



45

## Creating Distributed Applications with Sun RPC Example: Remote Dictionary Using `rpcgen`

- Procedure call structure:




Procedures should execute on the same machines as their resources are located.

46

## Specification for `rpcgen`

Specify:

- constants
- data types
- remote programs, their procedures, types of parameters

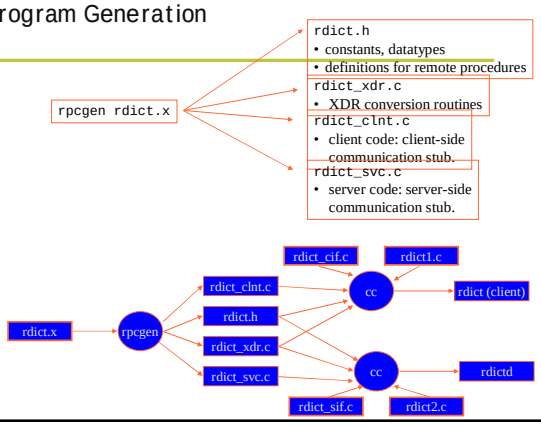
```
/* rdict.x */
/* RPC declarations for dictionary program */
const MAXWORD = 50;
const DICTSIZ = 100;
struct example { /* unused; rpcgen would      */
  int  exfield1; /* generate XDR routines     */
  char exfield2; /* to convert this structure.*/
};

/* RDICTPROG: remote program that provides
   insert, delete, and lookup */

program RDICTPROG {         /* name (not used) */
  version RDICTVERS {       /* version declarat.*/
    int INITW(void)     = 1;/* first procedure */
    int INSERTW(string)= 2;/* second proc.... */
    int DELETEW(string)= 3;
    int LOOKUP(string) = 4;
  } = 1;                    /* version definit.*/
} = 0x30090949;             /* program no      */
                            /* (must be unique)*/
```

47

## Program Generation




48

## Any Questions?

See you next time.