CprE 450/550X
Distributed Systems and Middleware

Inter-process Communication

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

Feb. 4&6, 2003

The First Course Project

- Distributed Chat Services – Based on RPC
- Due: 5:00pm, March 3, 2003
- Two students in a group
- More detailed please check the handout.

Readings for Today's Lecture

- References
  - **Chapter 2 of "Distributed Systems: Principles and Paradigms"**

Object-Oriented Distributed Technology

- Objects
- Objects in Distributed Systems
- Requirements of Multi-User Applications

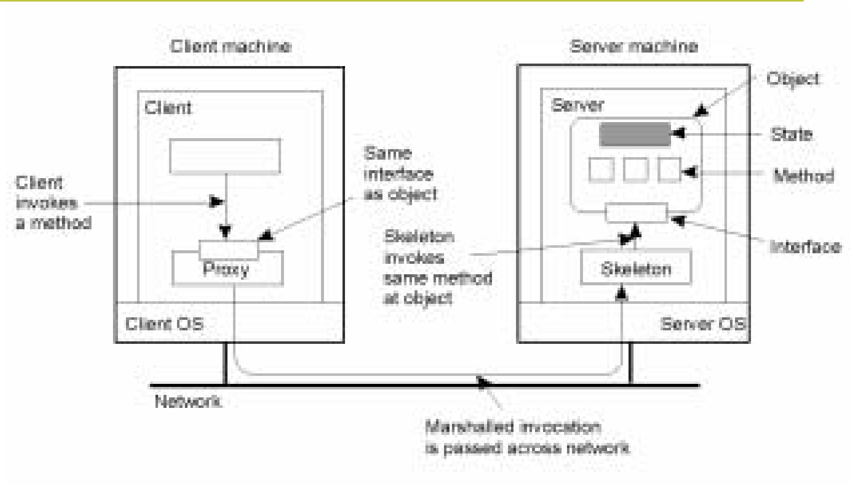
Object-Oriented Languages

- Object Identity
  - **“object identifiers” (OIDs)**
  - **OIDs as first class values**
- Actions
  - **Inititiated by sending message to object requesting method invocation**
  - **State in object may change**
  - **cascaded invocations of methods**
- Dynamic Binding
  - **The method executed is chosen according to the class of the recipient of the message.**
- Garbage Collection
  - **Dynamically allocated instances may be explicitely *deleted* or space is freed implicitely by garbage collector.**

Objects in Distributed Systems

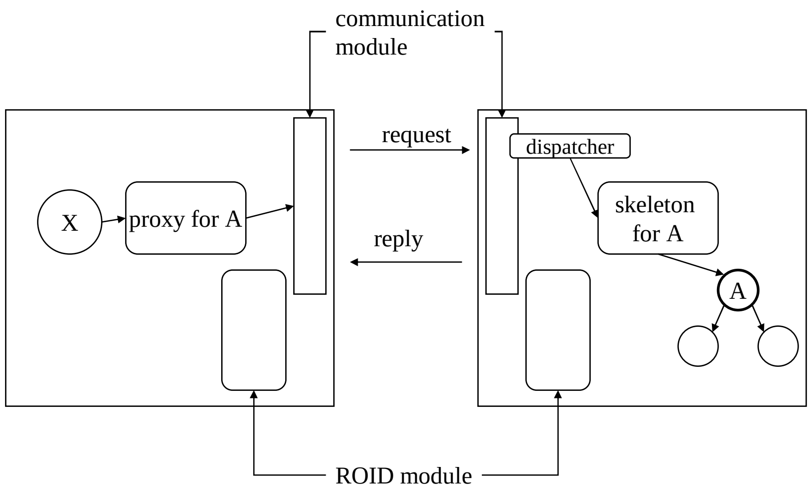
- Object Identity in a Distributed System
  - **Remote object identifiers (ROIDs)**
  - **Ex. Java: ROID = endpoint (Java vm) + identifier (ObjID)**
  - **ROIDs as first-class values**
  - **Service for comparing remote object identifiers**
    - **e.g. Java: *RemoteObject::equals()***
- Actions in a Distributed Object System
  - **Remote Method Invocation**
- The Role of Proxies for Transparent RMI
  - **Local <u>proxy</u> for each remote object that can be invoked by local object.**
  - **Local proxy behaves like local object, but, instead of executing message, forwards it to the remote object. (client stubs)**
  - **Remote object has <u>skeleton</u> object with server stub procedures**

Distributed Objects



- Common organization of a remote object with client-side proxy.

Proxies and Skeletons



Proxies and Skeletons (cont)

- Proxies:
  - **Need proxies to invoke remote objects.**
  - **Proxies are created when needed whenever ROID arrives in Reply message.**
  - **ROID module manages proxies and ROIDs.**

- Dispatchers and Skeletons:
  - **Not necessary for systems with reflection capabilities.**
  - **e.g. class *Method* in Java 1.2 reflection package:**
    - **method *invoke* can be called on instance of *Method*.**
    - **Dispatcher now generic and skeleton unnecessary.**

Binding a Client to an Object

```
Distr_object* obj_ref;              //Declare a systemwide object reference
obj_ref = …;                        // Initialize the reference to a distributed object
obj_ref-> do_something();           // Implicitly bind and invoke a method
```

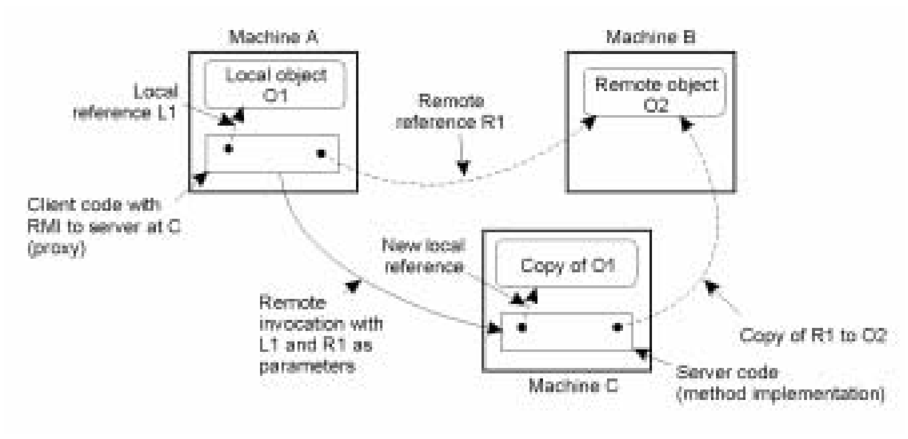
(a)

```
Distr_object objPref;               //Declare a systemwide object reference
Local_object* obj_ptr;              //Declare a pointer to local objects
obj_ref = …;                        //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);            //Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();          //Invoke a method on the local proxy
```

(b)

(a) Example with implicit binding using only global references
(b) Example with explicit binding using global and local references

Parameter Passing



- The situation when passing an object by reference or by value.

Arguments and Results in RMI

- Semantics of passing arguments for RMI in object-oriented languages needs to be defined.
- Argument and Result passing in Java RMI:
  - **When type of parameter is defined as remote interface, argument or result is passed as ROID (*by reference*).**
  - **Other non-remote objects may be passed *by value* if they are *serializable*.**
- Which objects can be accessed by RMI?
  - **Any object can be accessed by RMI**
  - **Distinguish between remote objects and local objects. (e.g. Java)**
  - **Use interface definition language (IDL)**
- Problem: migration/replication

Dynamic Binding

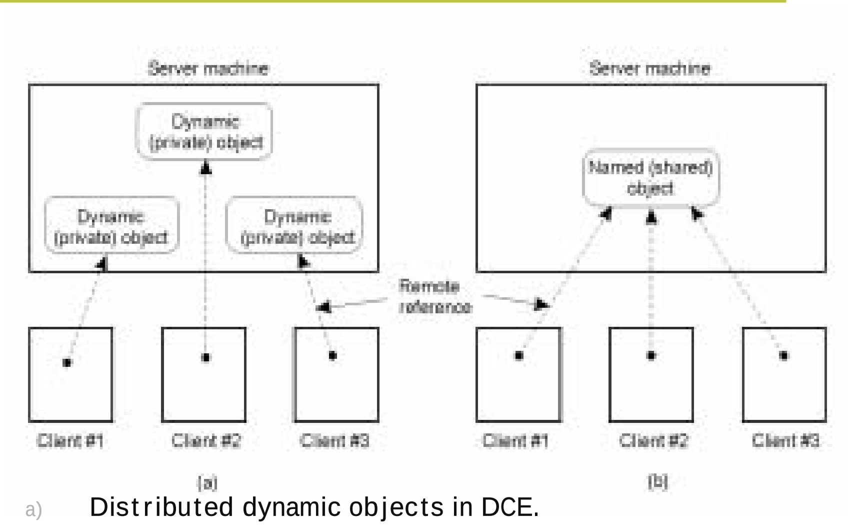
- Dynamic method binding should also apply to RMI.
- Smalltalk: Allow any message to be sent to any object, and raise exception if method is not supported.
  - **Distributed Smalltalk: general-purpose proxies.**
- Java RMI:
  - **dynamic binding as a natural extension of local case**
  - **Example:**

```
Shape aShape = (Shape) stack.pop();
float f = aShape.perimeter();
```

Garbage Collection

- Some languages (Java, Smalltalk) support garbage collection.
- Explicit memory management difficult/impossible in distributed environment.
- Distributed garbage collection typically realized in ROID modules. Each ROID module:
  - **keeps track how many sites hold remote ROIDs for each local object**
    (maintains `holders` table)
  - **informs other ROID modules about generation/deletion of ROIDs for their local objects (through the use of `addRef()` and `removeRef()`)**
- Local garbage collector collects objects with no local or remote references.
- Reference counting (`addROID()`/`removeROID()`) over unreliable networks

The DCE Distributed-Object Model



a) Distributed dynamic objects in DCE.
b) Distributed named objects

Remote Procedure Call (RPC)

Example (local procedure)

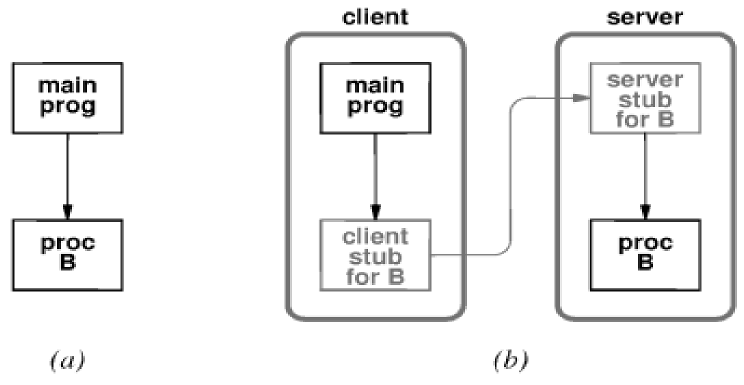
```
int f(x,y) {
    x = x+1; y = y+2;
    return(x+y);
}

main( )
{
    int a,b,c; a = 0; b = 1;
    c = f(a,b);
    printf("a = %d, b = %d, c = %d \n", a,b,c);
}
```

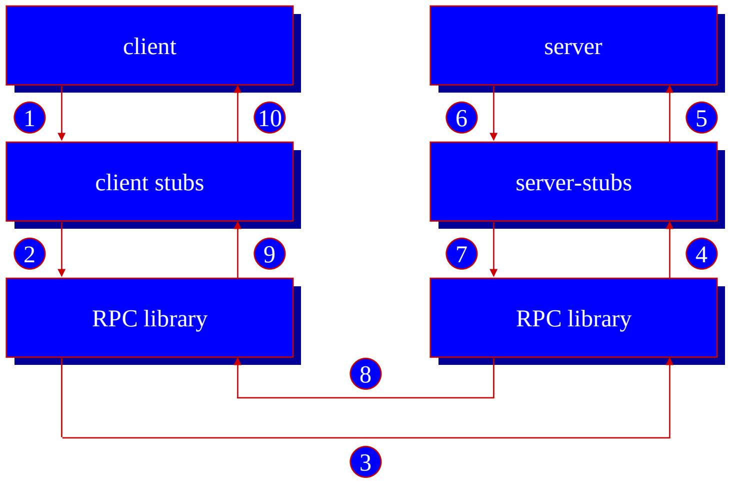
Parameter Passing Mechanism and Value Printed:

- Call-by-value: a = 0, b = 1, c = 4
- Call-by-reference: a = 1, b = 3, c = 4
- Call-by-copy/restore: a = 1, b = 3, c = 4

Client and Server Stubs



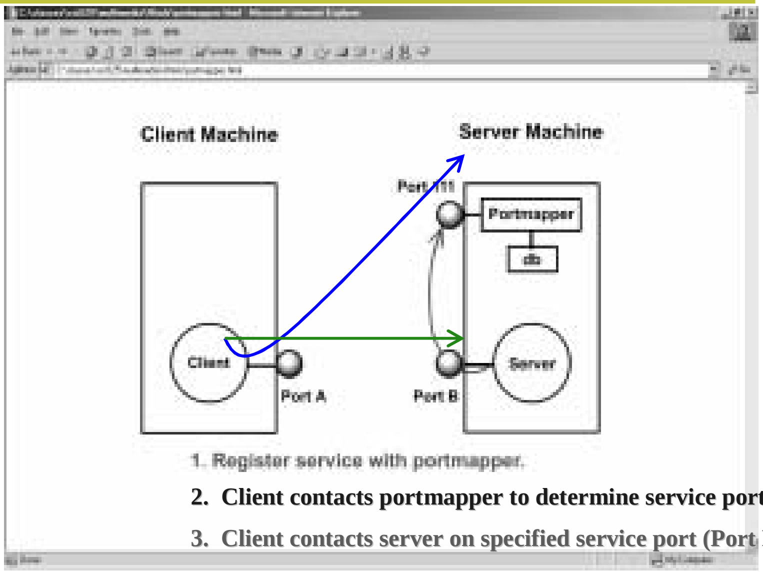
Structure of an RPC Call



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 Interaction



1. Register service with portmapper.
2. **Client contacts portmapper to determine service port.**
3. Client contacts server on specified service port (Port B).

RPC Implementation

Establishing an RPC Session

1. The server **registers** its services (procedures) with the portmapper.
2. The client **contacts the portmapper** to determine if the requested service (procedure) is available; and if so, on which port.
3. The client **contacts the server** to initiate service.

XDR - eXternal Data Representation

- **XDR** is a universally used standard from Sun Microsystems used to represent data in a network canonical form.
- A set of conversion functions are used to encode and decode data; for example, **xdr_int( )** is used to encode and decode integers. Data is converted into a network canonical form (a standard form) to be presented in a meaningful format to the receiving host.
- Conversion functions exist for all standard data types. However, for complex structures, RPCGEN can be used to generate conversion routines.

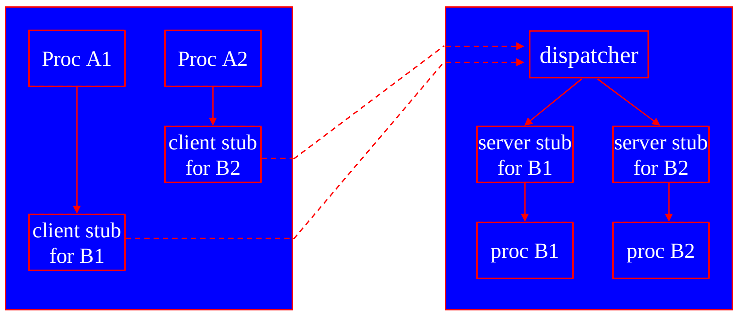
Sun RPC Message Format: XDR Specification

```
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;
};
```

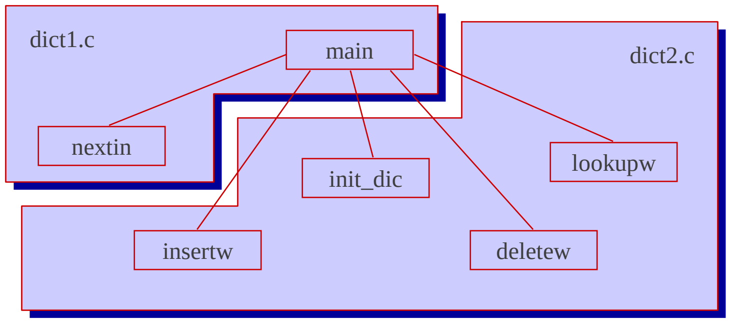
```
struct call_body {  /* format of RPC CALL                */
  u_int rpcvers;    /* which version of RPC?             */
  u_int rprog;      /* remote program number             */
  u_int rprogvers;  /* version number of remote prog */
  u_int rproc;      /* number of remote procedure        */
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

Message Dispatch for Remote Programs



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.

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)*/
```

Program Generation

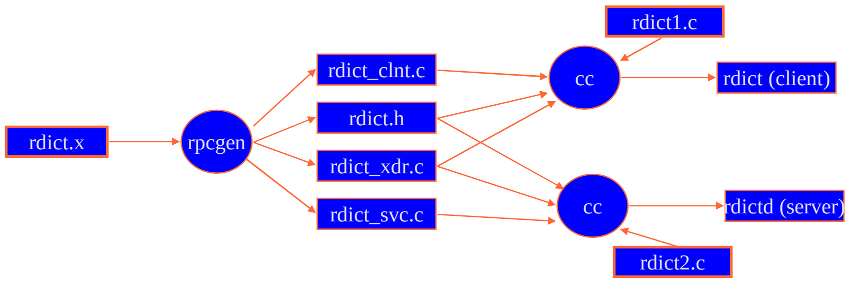
`rpcgen rdict.x`

`rdict.h`
- constants, datatypes
- definitions for remote procedures

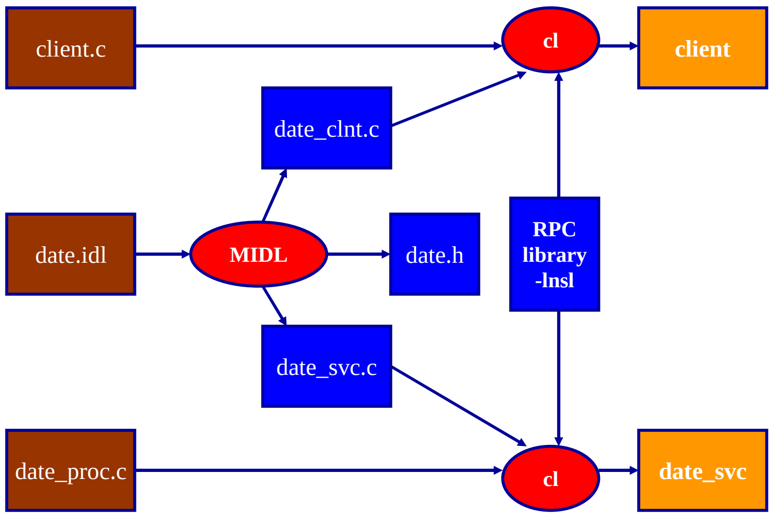
`rdict_xdr.c`
- XDR conversion routines

`rdict_clnt.c`
- client code: client-side communication stub.

`rdict_svc.c`
- server code: server-side communication stub.



Windows (DCE) RPC Example



Date.x

```
/*
 * date.x - Specification of remote date, time, date and time service.
 */

/*
 * Define 1 procedure :
 * date_1() accepts a long and returns a string.
 */

program DATE_PROG {
    version DATE_VERS {
            string    DATE(long) = 1;     /* procedure number = 1 */
    } = 1;                            /* version number = 1   */
} = 0x31234567;                          /* program number       */
```

Client.c

```
main(int argc, char **argv)
{
   CLIENT  *cl;       /* RPC handle */
   char    *server;
   char    **sresult; /* return value from date_1() */
   char    s[MAX];    /* character array to hold output */
   long    response;  /* user response                 */
   long    *lresult;  /* pointer to user response       */

   if (argc != 2) {
      fprintf(stderr, "usage: %s hostname\n", argv[0]);
      exit(1);
   }
   server = argv[1];
   lresult = (&response);
   /*
    * Create the client "handle."
    */
   if ( (cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
      clnt_pcreateerror(server);
      exit(2);
   }
   response = get_response();
   while(response != 4) {
      if ((sresult = date_1(lresult, cl)) == NULL) {
         clnt_perror(cl, server);
         exit(3);
      }
            printf("  %s\n", *sresult);
            response = get_response();
   }
   clnt_destroy(cl);                        /* done with the handle */
   exit(0);
}
```

33

Client.c (cont.)

```
#include    <stdio.h>
#include    <string.h>
#include    <time.h>
#include    <sys/types.h>
#include    <rpc/rpc.h> /* standard RPC include file */
#include    "date.h"     /* this file is generated by rpcgen */

#define MAX 100
long get_response(void);


long get_response()
{
    long choice;

    printf("===========================================\n");
    printf("                   Menu: \n");
    printf("-------------------------------------------\n");
    printf("                1. Date\n");
    printf("                2. Time\n");
    printf("                3. Both\n");
    printf("                4. Quit\n");
    printf("-------------------------------------------\n");
    printf("               Choice (1-4):");
    scanf("%ld",&choice);
    printf("===========================================\n");
    return(choice);
}
```

34

Server.c

```
/*
 * date_proc.c - remote procedures; called by server stub.
 */
#include <rpc/rpc.h>            /* standard RPC include file */
#include <time.h>
#include <sys/types.h>
#include "date.h"               /* this file is generated by rpcgen */

#define MAX 100
/*
 * Return the binary date and time.
 */
char **
date_1(option)
long *option;
{
   struct tm *timeptr; /* Pointer to time structure     */
   time_t clock;       /* Clock value (in secs)          */
   static char *ptr;   /* Return string                  */
   static char err[] = "Invalid Response \0";
   static char s[MAX];

   clock = time(0);
   timeptr = localtime(&clock);

   switch(*option)
      {
      case 1: strftime(s,MAX,"%A, %B %d, %Y",timeptr);
              ptr=s;
              break;

      case 2: strftime(s,MAX,"%T",timeptr);
              ptr=s;
              break;

      case 3: strftime(s,MAX,"%A, %B %d, %Y - %T",timeptr);
              ptr=s;
              break;

      default: ptr=err;
               break;
      }
   return(&ptr);
}
```