Distributed Computing

Remote Procedure Calls (RPC)

Dr. Yingwu Zhu

Problems with Sockets

- Sockets interface is straightforward
  - [connect]
  - read/write
  - [disconnect]
- BUT … it forces read/write mechanism
  - We usually use a procedure call
- To make distributed computing look more like centralized:
  - I/O is not the way to go

RPC

- 1984, Birrell & Nelson
  - Mechanism to call procedures on other machines
- Goal:
  - It should appear to the programmer that a conventional (local) call is taking place

Big Question

How do conventional procedure calls work in programming languages?

Conventional Procedure Calls

- Machine instructions for call & return but the complier really makes the procedure call abstract work:
  - Parameter passing
  - Local variables
  - Return data

Conventional Procedure Calls

You write: x = fun(a, “test”, 5);

The complier parses this and generate code to

1. Push 5 on the stack
2. Push the addr. of “test” on the stack
3. Push the current value of a on the stack
4. Generate a call to the function f

In compiling f, the compiler generates code to:

1. Push the registers that will be clobbered on the stack to save the values
2. Adjust the stack to make room for local and temporal variables (also return address)
3. Before a return, unadjust the stack (undo step 2), put the return data in a register, and issue a return instruction.

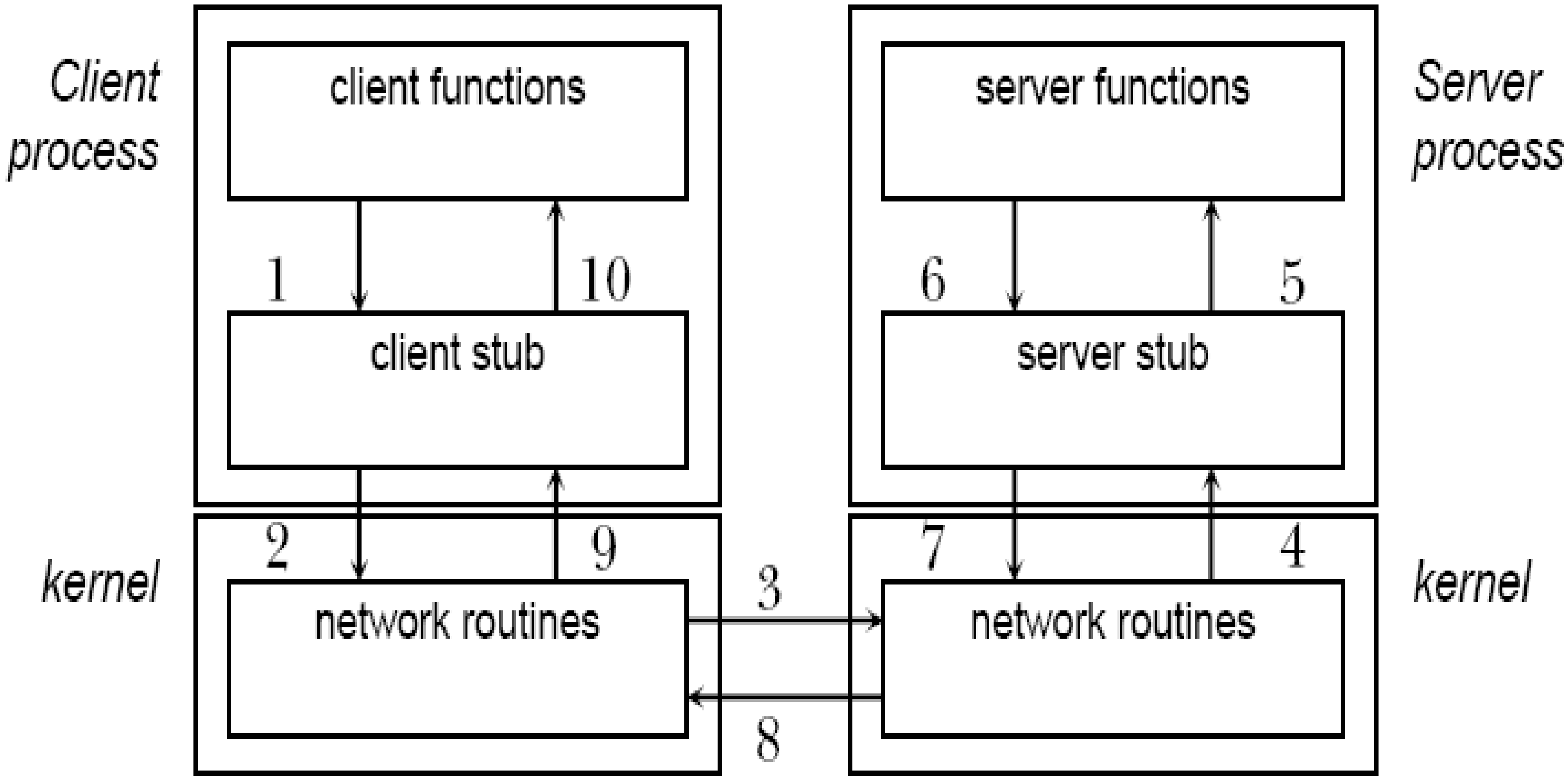
Implementing RPC

- No architectural support for RPC
- Simulate RPC with tools we have (local procedure calls)
  - Simulation make RPC a language-level construct
  - Instead of operating system construct

Implementing RPC

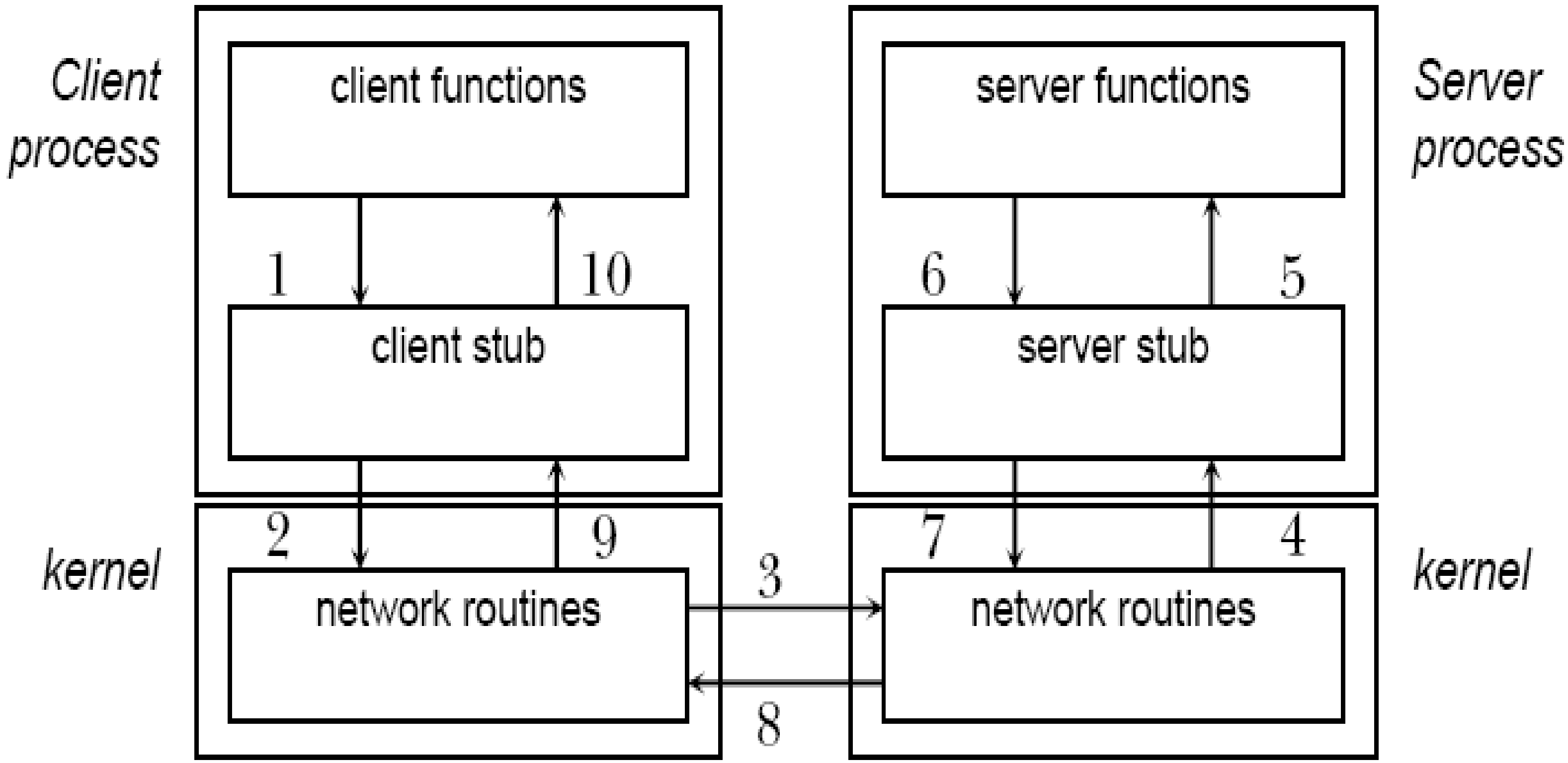
- The trick
  - Create stub functions to make it appear to the user that the call is local
  - Stub function contains the function's interface

Steps in RPC



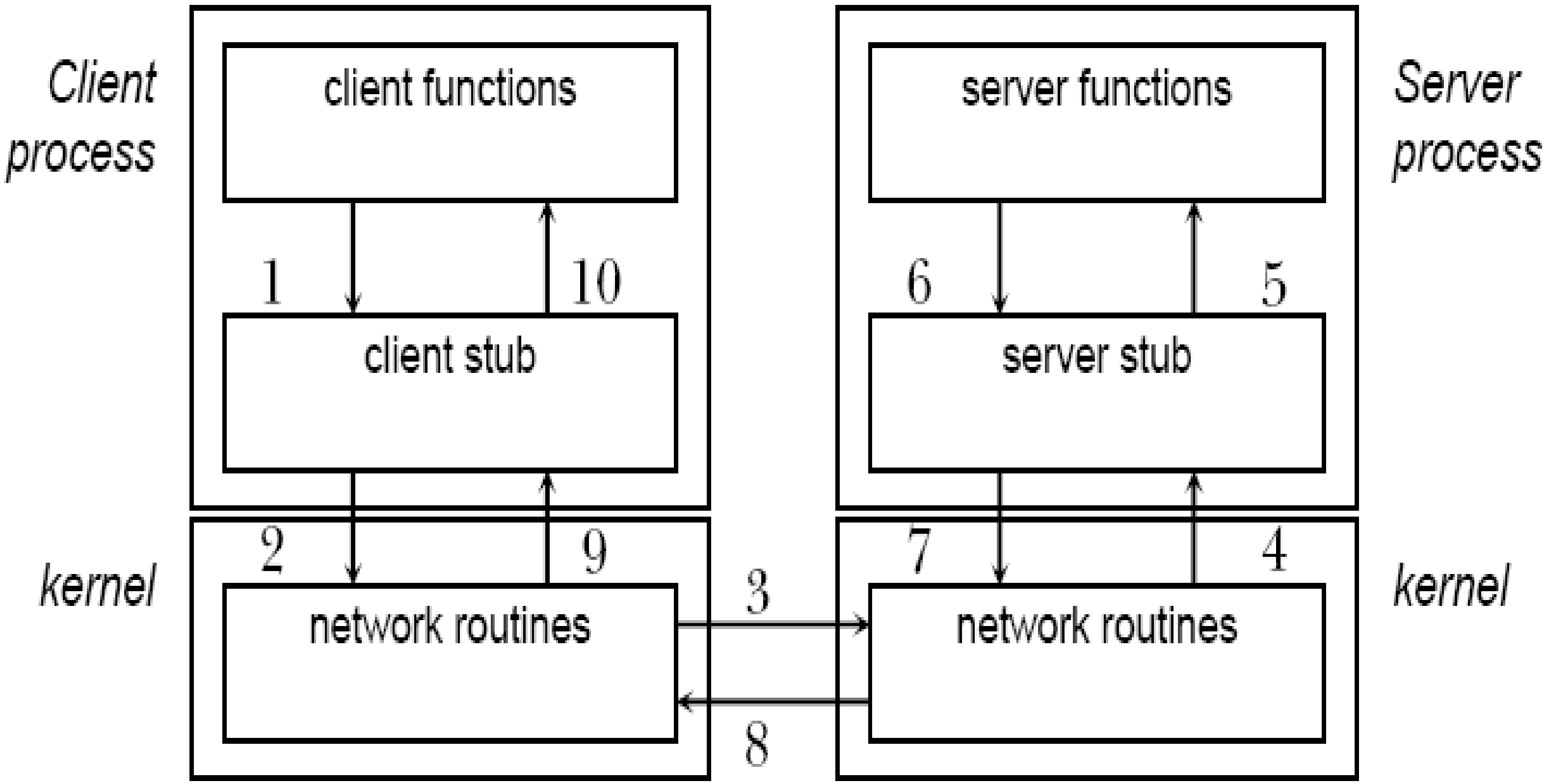
Step 1. The client calls the client stub which packaging the arguments into a network message (marshaling)
Step 2. The network message is forwarded to the network routines (local kernel by system call)

Steps in RPC



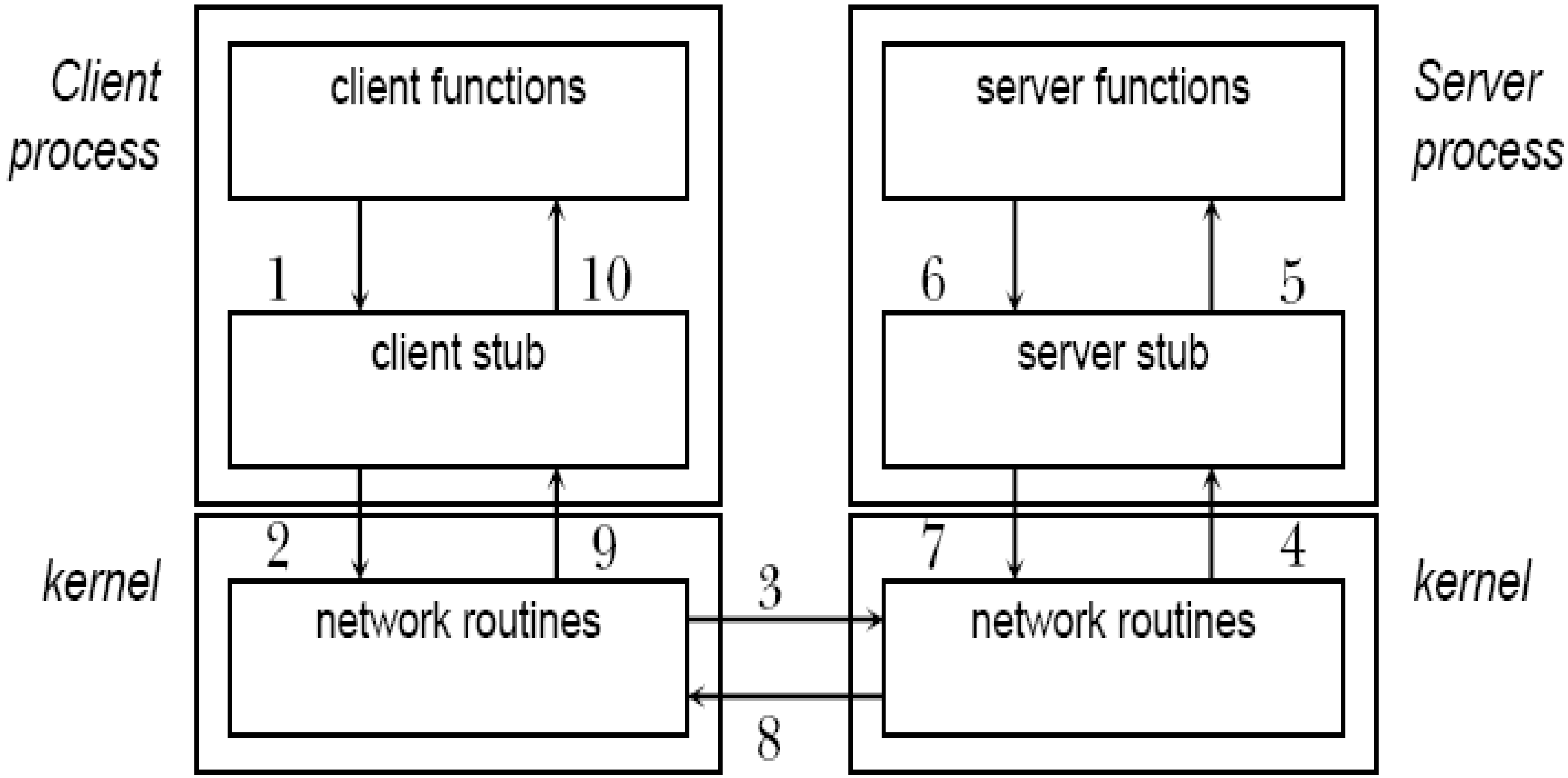
Step 3. The network message is sent to the remote server via some protocol (TCP/UDP)
Step 4. The sever stub unmarshals the arguments from the message & convert them into machine-specific format (big-endian, little-endian)

Steps in RPC



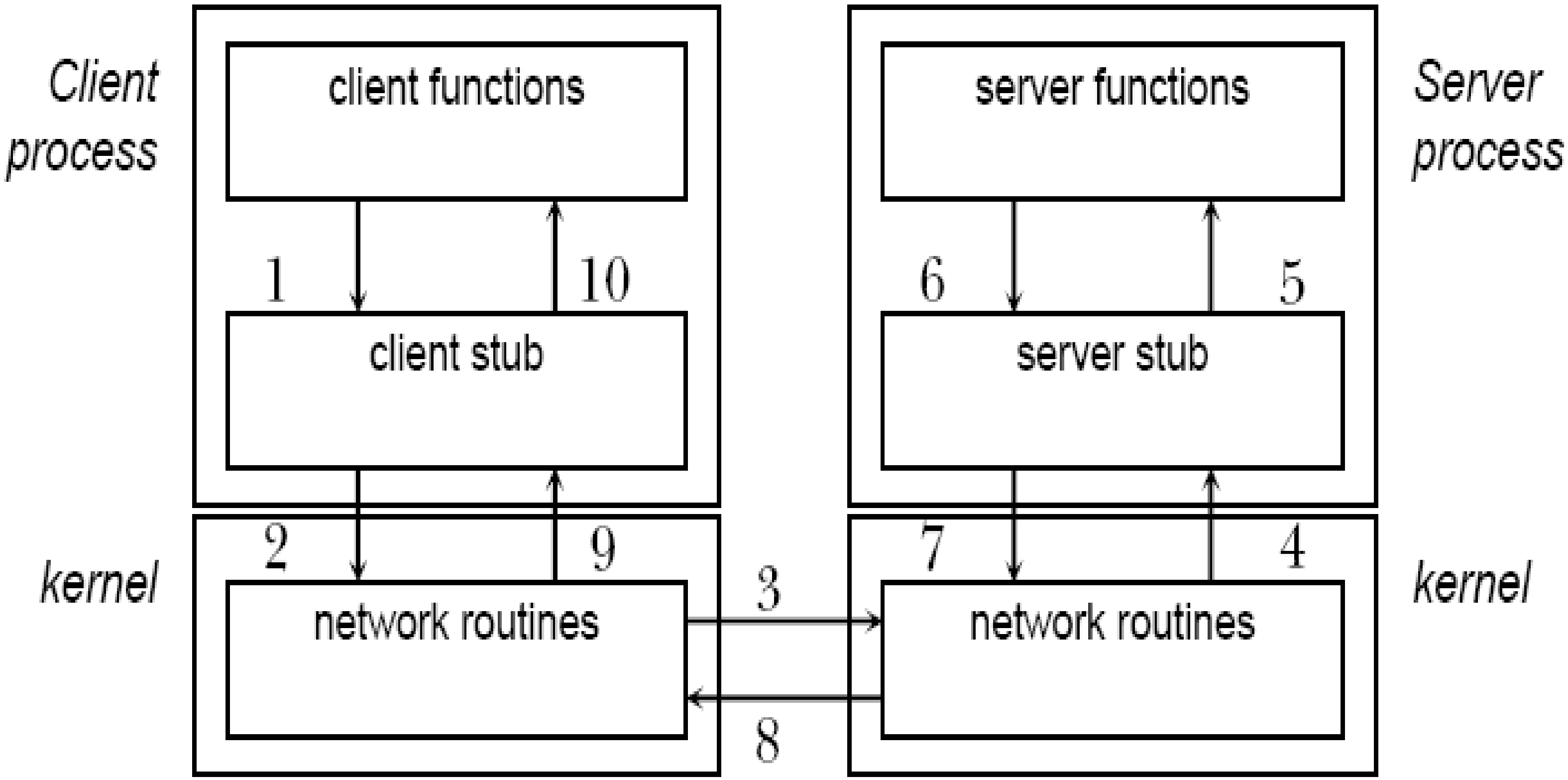
Step 5. The sever stub executes a local procedure call to the actual server function, by passing the arguments received
Step 6. When the call completes, it returns values back to the server stub

Steps in RPC



Step 7. The sever stub converts the return values (if necessary) and marshals them into one or more network messages to send to the client stub
Step 8. Messages are sent to the client stub

Steps in RPC



Step 9. The client stub reads the message from the local kernel

Step 10. It returns the results back to the client (possibly convert them first!)

RPC: Benefits

- Procedure call interface: we are familiar
- Writing applications is simplified
  - RPC hides all network code into stub functions
  - Application programmers don’t have to worry about details
    - Sockets, port numbers, byte ordering

RPC: Issues

Parameter passing

- Pass by value
  – Easy, just copy data into network messages
- Pass by reference
  – Make no sense without shared memory

Pass by reference?

How

1. Copy items referenced to message buffer
2. Ship them over
3. Unmarshal data at server
4. Pass local pointer to server stub function
5. Send new values back

To support complex structures

- Copy structure into pointerless representation
- Transmit
- Reconstruct structure with local pointers on server

Representing Data

- On local system: no *incompatibility* problems
- Remote machine may have
  - Different byte ordering
  - Different sizes of integers and other types
  - Different float point representations
  - Different character sets
  - Alignment requirement

Representing Data

IP (headers) forced all to use **big endian** byte ordering for 16 and 32 bit values

- *Most significant byte in low memory*
  - *Sparc, 680x0, MIPS, PowerPC G5*
  - Intel I-32 (x86/Pentium) use little endian

```
main() {
    unsigned int n;
    char *a = (char *)&n;

    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x\n",
           a[0], a[1], a[2], a[3]);
}
```

Output on a Pentium:
44, 33, 22, 11

Output on a PowerPC:
11, 22, 33, 44

Representing Data

- Need standard encoding to enable communication between heterogeneous systems
  - Sun’s RPC uses XDR (eXternal Data Representation)
  - ASN.1 (ISO Abstract Syntax Notation)

Representing Data

- Implicit typing
  - Only values are transmitted, no data types or parameter info
  - E.g., Sun XDR
- Explicit typing
  - Type is transmitted with each value
  - E.g., ISO's ASN.1, XML

Where to bind

- Need to locate host and correct server process
- Solution 1
  - Maintain centralized DB that can locate a host that provides a particular service (Birrel & Nelson 1984)
- Solution 2
  - A sever on each host maintains a DB of locally provided services

Transport protocols

- Some implementations may offer only one, e.g., TCP
- Most support several
  - Allow programmers to choose

When things go wrong

- Local procedure calls do not fail
  - If they core dump, entire process dies
- More opportunities for error with RPC
- Transparency breaks here!
  - Applications should be prepared to deal with RPC failures

When things go wrong

- Local procedure call: exactly once when we call it
- A remote procedure call may be called:
  - 0 times: server crashed or process died before executing server code
  - 1 time: every worked well
  - 1 or more: excess latency or lost reply from server and client retransmission

RPC Semantics

- Most RPC systems will offer either
  - At least once semantics
  - Or at most once semantics
- Understand application:
  - idempotent functions: may be run any number of times without harm (e.g., return date)
  - Non-idempotent functions: side effects
    - Modify or append a file

More issues

- Performance
  - RPC is slower… a lot slower than local procedure call
- Security
  - Messages visible over network
  - Authenticate client
  - Authenticate server

Programming with RPC

- Language support
  - Most programming languages (C/C++, Java,…) have no concept of remote procedure calls
  - Language compliers will not generate client and server stubs

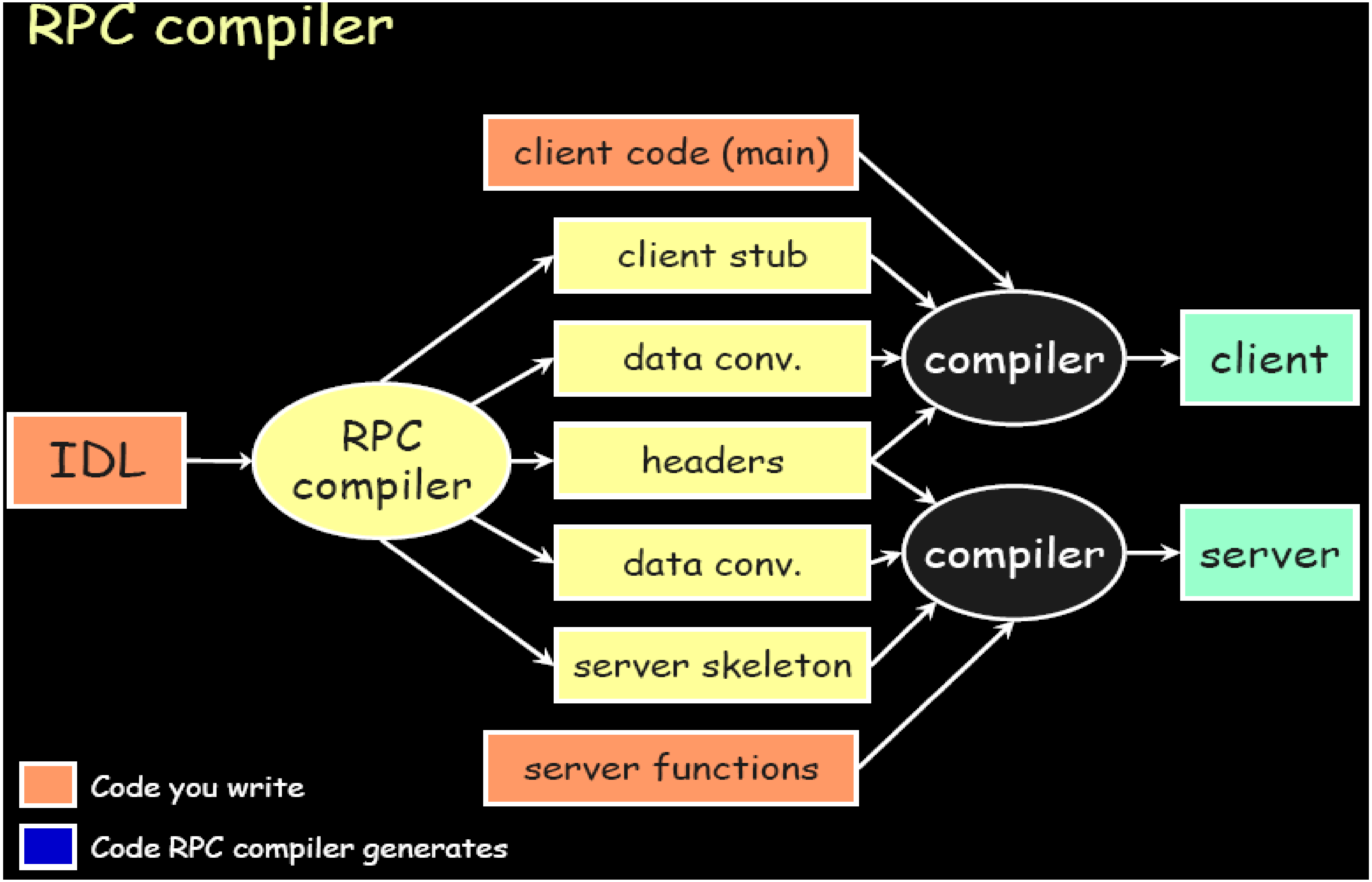
Common solution:

  - Use a separate complier to generate stubs (pre-complier: rpcgen)

Interface Definition Language (IDL)

- Allow programmers to specify remote procedure interfaces
  (names, parameters, return values)
- Pre-compiler can use this to generate client and server stubs:
  - Marshaling code
  - Unmarshaling code
  - Network transport routines
  - Conform to defined interface
- Similar to function prototypes

RPC Complier



Write the Program

- Client code has to be modified
  - Initialized RPC-related options
    - Transport type
    - Local server/service
  - Handle failures of remote procedure calls
- Server functions
  - Generally need little or no modification

RPC API

What kind of services does a RPC system need?

- Name service operations
  - Export/lookup binding info. (ports, machines)
  - Support dynamic ports
- Binding operations
  - Establishing c/s communication using appropriate protocol (establish endpoints)
- Endpoint operations
  - Listen for requests, export endpoint to name server

RPC API

What kind of services does a RPC system need?

- Security operations
  - Authenticate C/S
- Marshaling/Data conversion operations
- Stub memory management
  - Dealing with “reference” data, temporary buffers
- Program ID operations
  - Allow app. to access IDs of RPC interfaces

Case Study: SUN RPC

- RPC for Unix System V, Linux, BSD, OS X
  - Also known as ONC RPC(Open Network Computing)
- Interfaces defined in an Interface Definition Language (IDL)
  - IDL compiler is **rpcgen**

IDL

Interface Definition Language

- Used by *rpcgen* to generate stub functions
- defines an RPC *program*: collection of RPC procedures
- structure:

```
type definitions

program identifier {
        version version_id {
                procedure list
        } = value;
        ...
} = value;
```

```
program PROG {
        version PROG1 {
                void PROC_A(int) = 1;
        } = 1;
} = 0x3a3afeeb;
```

IDL Program: RPC interfaces

- Each IDL program contains the following structure:
  - optional constant definitions and typedefs may be present
  - the entire *interface is enveloped in a* ***program block.***
  - within the program block, one or more sets of *versions may be defined,* {program#, version#} tuple
  - within each version block, a set of functions is defined

Each collection of RPC interfaces is defined by a 32-bit value

- Unique value
  - 0x00000000-0x1fffffff: defined by sun
  - 0x20000000-0x3fffffff: defined by the user
  - 0x40000000-0x5fffffff: transient processes
  - 0x60000000-0x7fffffff: reserved

Data Types

Data types

- constants
  - may be used in place of an integer value - converted to #define statement by *rpcgen*

```
const MAXSIZE = 512;
```

- structures
  - similar to C structures - *rpcgen* transfers structure definition and adds a typedef for the name of the structure

```
struct intpair { int a, b };
```

is translated to:

```
struct intpair { int a, b };
typedef struct intpair intpair;
```

Data types

- **enumerations**
  - similar to C

```
enum state { BUSY=1, IDLE=2, TRANSIT=3 };
```

- **unions**
  - not like C
  - a union is a specification of data types based on some criteria:

```
union identifier switch (declaration) {
        case_list
}
```

  - for example:

```
const MAXBUF=30;
union time_results switch (int status) {
        case 0: char timeval[MAXBUF];
        case 1: void;
        case 2: int reason;
}
```

Data types

- type definitions
  - like C:
    ```
    typedef long counter;
    ```
- arrays
  - like C but may have a fixed or variable length:
    ```
    int proc_hits[100];
    ```
    defines a fixed size array of 100 integers.

    ```
    long x_vals<50>
    ```
    defines a variable-size array of a maximum of 50 longs
- pointers
  - like C, but nit sent over the network. What is sent is a boolean value (true for pointer, false for null) followed by the data to which the pointer points.

Data types

- **strings**
  - declared as if they were variable length arrays
    ```
    string name<50>;
    ```
    declares a string of at most 50 characters.
    ```
    string anyname<>;
    ```
    declares a string of any number of characters.
- **boolean**
  - can have the value of TRUE or FALSE:
    ```
    bool busy;
    ```
- **opaque data**
  - untyped data that contains an arbitrary sequence of bytes - may be fixed or variable length:
    ```
    opaque extra_bytes[512];
    opaque more<512>;
    ```
  - latter definition is translated to C as:
    ```
    struct {
            uint more_len;    /* length of array */
            char *more_val;   /* space used by array */
    }
    ```

Writing procedures using Sun RPC

- create a procedure whose name is the name of the RPC definition
  - in lowercase
  - followed by an underscore, version number, underscore, “svc”
  - for example, BLIP → blip_1_svc
- argument to procedure is a *pointer to the argument data type specified in* the IDL
- default behavior: only *one parameter to each function*
  - if you want more, use a struct
  - this was relaxed in later versions of rpcgen but remains the default
- procedure must return a *pointer to the data type specified in the IDL*
- the server stub uses the procedure’s return value after the procedure returns, so the return address must be that of a **static variable**

Step-by-Step for a RPC program

Step by Step for a RPC program

- Start with stand-alone program that has two functions:
  - bin_date returns system date as # seconds since Jan 1 1970 0:00 GMT
  - str_date takes the # of seconds as input and returns a formatted data string
- Goal
  - move bin_date and str_date into server functions and call them via RPC.

Standalone program

```
#include <stdio.h>

long bin_date(void);
char *str_date(long bintime);

main(int argc, char **argv) {
        long lresult; /* return from bin_date */
        char *sresult; /* return from str_date */
        if (argc != 1) {
                fprintf(stderr, "usage: %s\n", argv[0]
                exit(1);
        }
        /* call the procedure bin_date */
        lresult = bin_date();
        printf("time is %ld\n", lresult);
        /* convert the result to a date string */
        sresult = str_date(lresult);
        printf("date is %s", sresult);
        exit(0);
}
```

```
/* bin_date returns the system time in binary format */
long bin_date(void) {
        long timeval;
        long time();  /* Unix time function; returns time */

        timeval = time((long *)0);
        return timeval;
}


/* str_date converts a binary time into a date string */
char *str_date(long bintime) {
        char *ptr;
        char *ctime();  /* Unix library function that does the work */

        ptr = ctime(&bintime);
        return ptr;
}
```

Step 1: IDL to define interface

- Define two functions that run on server:
  - bin_date has no input parameters and returns a long.
  - str_date accepts a long as input and returns a string
- IDL:

```
program DATE_PROG {
      version DATE_VERS {
            long BIN_DATE(void) = 1;
            string STR_DATE(long) = 2;
      } = 1;
} = 0x31423456;
```

function numbers (→ 1, 2)

version number (→ `} = 1;`)

program number (→ `} = 0x31423456;`)

- IDL convention is to suffix the file with .x
  - we name the file date.x
  - it can be compiled with:

```
rpcgen -C date.x
```

Step 2: Pre-complier

rpcgen -C date.x

– -C is to produce ANSI C function declarations

We get

- date_clnt.c : client stub
- date.h : header file
- date_svc.c : server stub
- date_xdr.c: (possible)

Step 3: Generate server functions: template from rpcgen

We can have *rpcgen* generate a template for the server code using the interface we defined:

```
rpcgen -C -Ss date.x >server.c
```

This produces:

```
#include "date.h"
long *
bin_date_1_svc(void *argp, struct svc_req *rqstp)
{
	static long result;
	/* insert server code here */
	return &result;
}

char **
str_date_1_svc(long *argp, struct svc_req *rqstp)
{
	static char *result;
	/* insert server code here */
	return &result;
}
```

Step 4: plug the server function code

Now just copy the functions from the original stand-alone code

```
long *
bin_date_1_svc(void *argp, struct svc_req *rqstp)
{
    static long result;
    long time();
    result = time((long *)0);
    return &result;
}

char **
str_date_1_svc(long *bintime, struct svc_req *rqstp)
{
    static char *result;
    char *ctime();

    result = ctime(bintime);
    return &result;
}
```

we don't need to use &bintime here because we get the address as a parameter

Step 5: generate the client code

rpcgen -C -Sc date.x > client.c

Modify the client code:

- Need to handle the server name
- Before we can make any remote procedure calls, we need to initialize the RPC connection via *clnt_create:*

```
CLIENT *cl; /* rpc handle */
cl = clnt_create(server, DATE_PROG, DATE_VERS, "netpath");
```

- "netpath" directs to read the NETPATH environment variable to decide on using TCP or UDP
- The server's RPC name server (port mapper) is contacted to find the port for the requested program/version/transport.
- Check for RPC errors for RPC calls! (if the pointer returned is null, then the call failed.)

Putting together: compile-link-run

- Generate stubs and client.c & server.c: **rpcgen -a -C date.x**
- Compile & link the client and client stub

  cc -o client client.c date_clnt.c -lnsl
- Compile & link the server and server stub

  cc -o server -DRPC_SVC_FG server.c date_svc.c -lnsl

  – Note: defining RPC_SVC_FG compiles the server such that it will run in the foreground instead of running as a background process
- Run the server (e.g. on remus)

  $ ./server
- Run the client

  $ ./client localhost
  time on localhost is 970457832
  date is Sun Oct 1 23:37:12 2000