# Grid Computing

## Paradigms for Distributed Computing 1
## Socket Programming & RPC

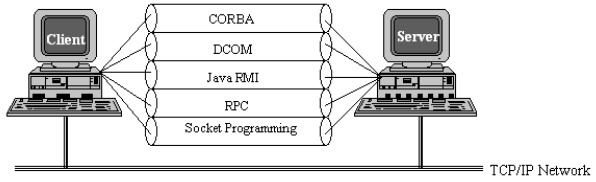
Paul A. Farrell
Fall 2006

The Grid: Core Technologies
Maozhen Li, Mark Baker
John Wiley & Sons; 2005, ISBN 0-470-09417-6

From material by Amy Apon at U. Arkansas



## Traditional paradigms for distributed computing






## Socket Programming

- Analogous to a two-way pipe
- Provide a low-level API for writing distributed client/server applications
- Socket end-points need to be created on the client and server
- Transport protocol TCP or UDP
- Client specifies host name and port
- Server listens on port
- Can “advertise” services on “well-known” ports
- Listed in /etc/services



## Socket Programming

- Advantages
  - API is well defined
  - can run on different operating systems
  - low latency and high-bandwidth
- Disadvantages
  - implementation is language dependent
  - multiple components interacting in complex ways require developer to explicitly create, maintain, manipulate, and close multiple sockets
- More information
  - http://www.cs.kent.edu/~farrell/sys2002/

## Remote Procedure Call

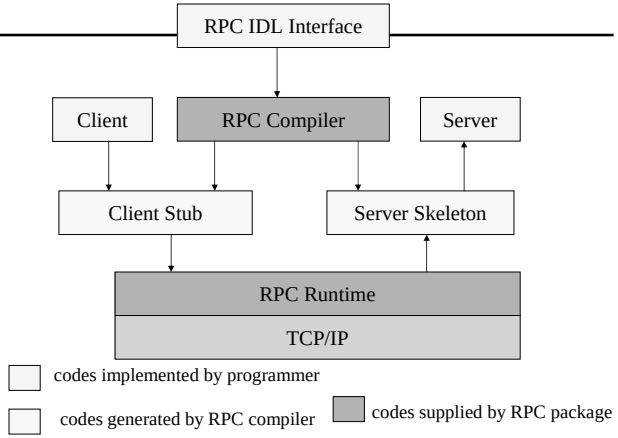
- RPC hides the low-level communication and provides a high-level communication abstraction
- Can use either TCP or UDP
- Uses Interface Definition Language (IDL) interface to describe the server-side remote procedures
- RPC compiler can automatically generate a client-side stub and a server-side skeleton from IDL
- RPC is a specification
- Various implementations
  - Open Network Computing (ONC) from Sun Microsystems
  - Distributed Computing Environment (DCE) from the Open Software Foundation (OSF)



## RPC

- RPC client and server have to be implemented in the same language and use the same RPC package
- Client needs to provide the server's host name
- Advantages
  - easier to use than sockets
- Disadvantages
  - Standard RPC only supports synchronous communication
  - Not object oriented








## Steps to Implement RPC

- Write a RPC interface in RPC IDL.
- Use a RPC compiler to compile the interface to generate a client-side stub and a server-side skeleton.
- Implement the server.
- Implement the client.
- Compile all the code with a RPC library.
- Start the server.
- Start the client with the IP address of the server.

## IDL Example

- IDL for a binary addition example.
- The call to the client stub in C from the application program is: binop_add(h, a, b, &)
- *h* is a handle to state information kept on the client. The sum is returned by means of a pointer to *c*.

```
[uuid(f9f6be80 2ba7 1c9 89fd 0002b13d56d),
version(0),
endpoint("ncadg_ip_udp:[6667]", "dda:[19]")]
interface binopwk
{

[idempotent] void binopwk_add
  (
    [in] handle_t h,
    [in] long a,
    [in] long b,
    [out] long *c
  )
}
```



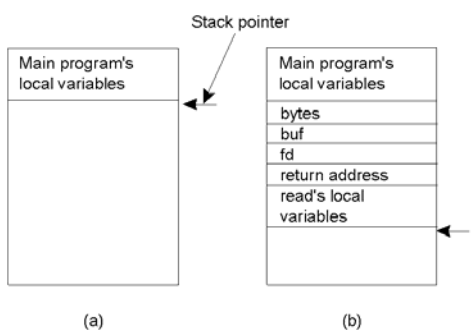
## Conventional Procedure Call Steps

- main() {
- …
- myretval = mysub(myparm);
- …
- }
- int mysub(int myparm) {
- /* calculate return value */
- return(returnvalue);

- steps:
- push myparm onto stack
- push return address onto stack
- branch to address of mysub
- /* allocate stack space for local vars */
- get myparm from stack, calculate ret val
- push return value onto stack
- /* fix stack */
- branch to return address
- get ret val from stack assign to myretval
- fix stack



## Conventional Procedure Call Steps




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



## Steps of RPC

- There is no stack or local shared memory with remote procedure call! Everything has to be done with message passing.
- With RPC
- steps: /* using local procedure call mechanisms where needed */
- main() {
- ...
- myretval = myrpcstub(myparm); /* local procedure call */
- ...
- }

## Steps of RPC – Client view

- int myrpcsub(int myparm) { /* client side view */

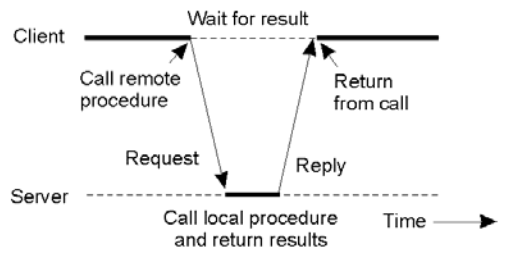
```
    /* calculate return value */
    pack myparm into a network-neutral format        /* serverstub */
    send message to myrpcserver with myparm--------> receive message
                                                     unpack parameter
                                                     retval = mysub(myparm)
                                                     pack retval into message
    receive return message  <---------------------- send retval back
    unpack return value from message
    return(returnvalue);
}
```

- Notice - you can't pass pointers, only pass by value



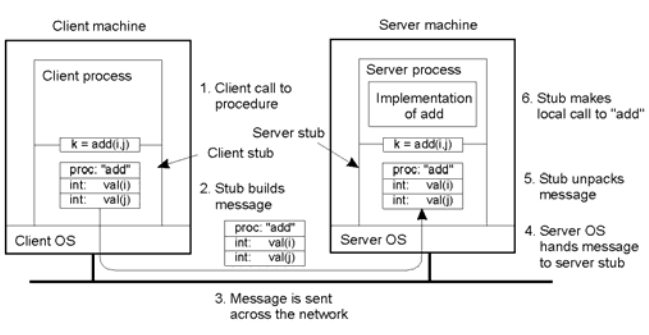
## Client and Server Stubs

- Principle of RPC between a client and server program.






## Steps of RPC

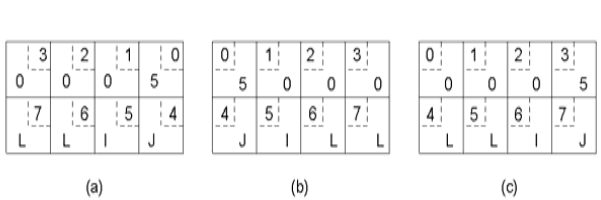





## Steps of a Remote Procedure Call

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 and numeric parameters




a) Original message on the Pentium
b) The message after receipt on the SPARC
c) The message after being inverted. The little numbers in boxes indicate the address of each byte
d) Parameters and return values are *marshalled* (packaged) into a network neutral format to fix endian and other format incompatibilities.

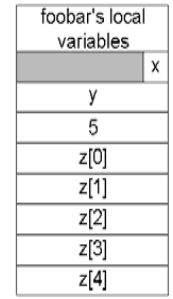


## RPC and array parameters

a) A procedure
b) The corresponding message.
c) In this example, the whole contents of the array is passed as a parameter!

```
foobar( char x; float y; int z[5] )
{
  ....
}
```

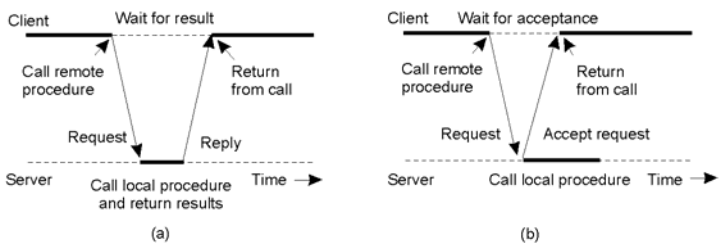
(a)




(b)



## Asynchronous RPC

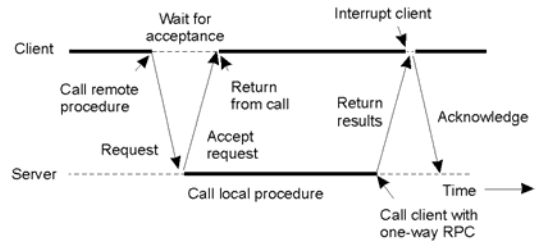



a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC



## Asynchronous RPC

- A client and server interacting through two asynchronous RPCs
- This is very similar to *notification* in Grid protocols



## RPC Semantics in presence of failure

- Some bad things that could happen:
  - The sent request could get lost (the server never executes it)
  - The server could crash and partially executes the request
  - The response could get lost (the server executes the whole request, but the network fails in some way)
  - The acknowledgement of the response by the client could get lost



## RPC Semantics in presence of failure

- Exactly once
  - The result is returned to the calling application exactly once
  - Requires that the client buffer the request until the response is received – have to number the requests so that you can tell them apart
  - Requires that the server buffer the result until the client confirms that the result has been received – have to also number the responses so that you can tell them apart
  - The server must ensure that the same request from the client is *idempotent* – the same request delivers the same response and has no additional affect on the system, even if it arrives more than one time
  - How long do you wait for a response??



## RPC Semantics in presence of failure

- At most once
  - The result is not guaranteed to be returned, but the same request will only return one time – can time out after a period of time and not return a result at all. How do you handle this in the calling program?
  - Still requires that the client buffer the request until the response is received, and number the requests. But how long do you buffer?
  - Still requires that the server buffer the result until the client confirms that the result has been received, and number the responses. But how long do you buffer?



## RPC Semantics in presence of failure

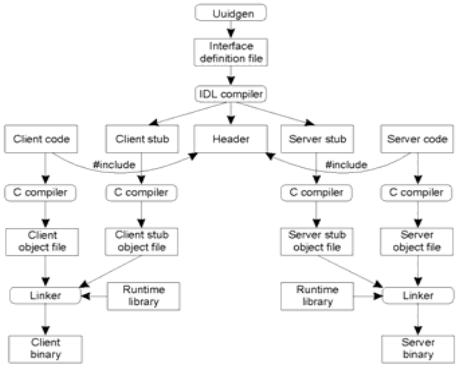
- At least once
  - The result is guaranteed to be returned, but the same request may return more than one time. How long do you wait?
  - Is it still idempotent?
  - Is buffering and numbering of requests required?
  - Is buffering and number of responses required?
- Another choice: one of many
  - Return any response that comes from the server

## How does the client locate the server?

- Could be located at compile time
  - The server address is embedded into the source
- Could be located at run time
  - Could be a command line parameter to the client program. The user needs to know the complete address of the server.
- Could be located using a lookup by name
  - The server registers a name into a name or directory service. The client must call the directory service before contacting the server for the first time to get its address
- Could be located using a lookup by type or capability
  - The server registers by capability into a directory service. The client calls the directory service before contacting the server for the first time. The client selects a server to use.



## Writing an RPC Client and a Server




- The steps in writing a client and a server in RPC.



## Binding a Client to a Server

- Client-to-server binding in DCE RPC