Week 4

Multiprocess Communication

Oana Balmau

January 26, 2023

Message Passing Alternatives

- Symmetric / asymmetric addressing
- Blocking / nonblocking

Symmetric Addressing

- **`Send`**`(msg, to_pid)`
- **`Receive`**`(msg, from_pid)`

- Message is (typically) a struct
- to_pid, from_pid are process identifiers

- Symmetric addressing seldom used

Asymmetric Addressing

- **`Send`**`(msg, pid)`
  - Send msg to process pid

- `pid =` **`Receive`**`(msg)`
  - Receive msg from *any* process
  - Return the pid of sending process

- More common and useful form of addressing

Blocking or Nonblocking Send

- Nonblocking:
  - Send returns immediately after message is sent
- Blocking
  - Sender blocks until message is delivered
- Nonblocking is the more common form

Blocking or Nonblocking Receive

- Nonblocking
  - Receive returns immediately
  - Regardless of message present or not
- Blocking
  - Receive blocks until message is present
- Blocking is the more common form

(Slightly Rewritten) Example: Multiprocess Web Server with Process Pool

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
       client_pid = receive(msg)
       msg' = slightly modify msg to include client_pid
       send(msg', worker_process[i])
    }
}

WorkerProcess[i] {
  forever {
    receive(msg)
    read file from disk
    send(resp, client_pid)
  }
}
```

Asymmetric Addressing: Send

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      client_pid = receive(msg)
      msg' = slightly modify msg to include client_pid
      send(msg', worker_process[i])
    }
}

WorkerProcess[i] {
  forever {
    receive(msg)
    read file from disk
    send(resp, client_pid)
  }
}
```

Asymmetric Addressing: Receive

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      client_pid = receive(msg) /* receive msg from any client */
      msg' = slightly modify msg to include client_pid
      send(msg', worker_process[i])
    }
}

WorkerProcess[i] {
  forever {
    receive(msg) /* receive msg' from listener; could be symmetric */
    read file from disk
    send(resp, client_pid)
  }
}
```

Blocking Receive

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      client_pid = receive(msg) /* nothing else to do*/
      msg' = slightly modify msg to include client_pid
      send(msg', worker_process[i])
    }
}

WorkerProcess[i] {
  forever {
    receive(msg) /* nothing else to do*/
    read file from disk
    send(resp, client_pid)
  }
}
```

Nonblocking Send

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      client_pid = receive(msg)
      msg' = slightly modify msg to include client_pid
      send(msg', worker_process[i]) /* must not block */
    }
}

WorkerProcess[i] {
  forever {
    receive(msg)
    read file from disk
    send(resp, client_pid) /* must not block */
  }
}
```

Client-Server Communication

(Server Side) Client-Server Communication

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      receive incoming request
      send( request, process[?] )
    }
}

WorkerProcess[?] {
  forever {
    wait for message( &request )
    read file from disk
    send response
  }
}
```

(Client-Side) Client-Server Communication

```
send(msg to server)

receive(reply msg from server)
```

A Very Common Pattern

- Client:
  - Send /* send request to server */
  - Blocking receive /* wait for reply */
- Server
  - Blocking receive /* wait for request */
  - Send /* send reply */

This looks like …

• Client:	calling site
• Send	call procedure
• Blocking receive	return
• Server	callee site
• Blocking receive	invoke procedure
• Send	return

Remote Procedure Call (RPC)

- Client:
  - Send
  - Blocking receive
- Server
  - Blocking receive
  - Send

calling site

call procedure

return

callee site

invoke procedure

return

RPC: when client wants to call a function that belongs to server code

RPC Interface

- Interface
  - List of remotely callable procedures
  - With their arguments and return values

- Example: file system interface
  - `Open(string filename)`
  - returns `int fd`
    - fd = file descriptor; will see later in course
  - …

RPC Client Code

- Import file system interface

- `fd = open("/a/b/c")`
- `nbytes = read(fd, buffer, size)`

RPC Server Code

- Export file system interface

- `int open(stringname) { … }`
- `int read(fd, buffer, nbytes) { … }`
- …

Problem

- Want a procedure call interface
- Have only message passing between processes
- How to bridge the gap?

Solution: Stub Library

- Client stub and server stub
- Client stub linked with client process
- Server stub linked with server process

Two Message Types

- Call message
  - From client to server
  - Contains arguments
- Return message
  - From server to client
  - Contains return values

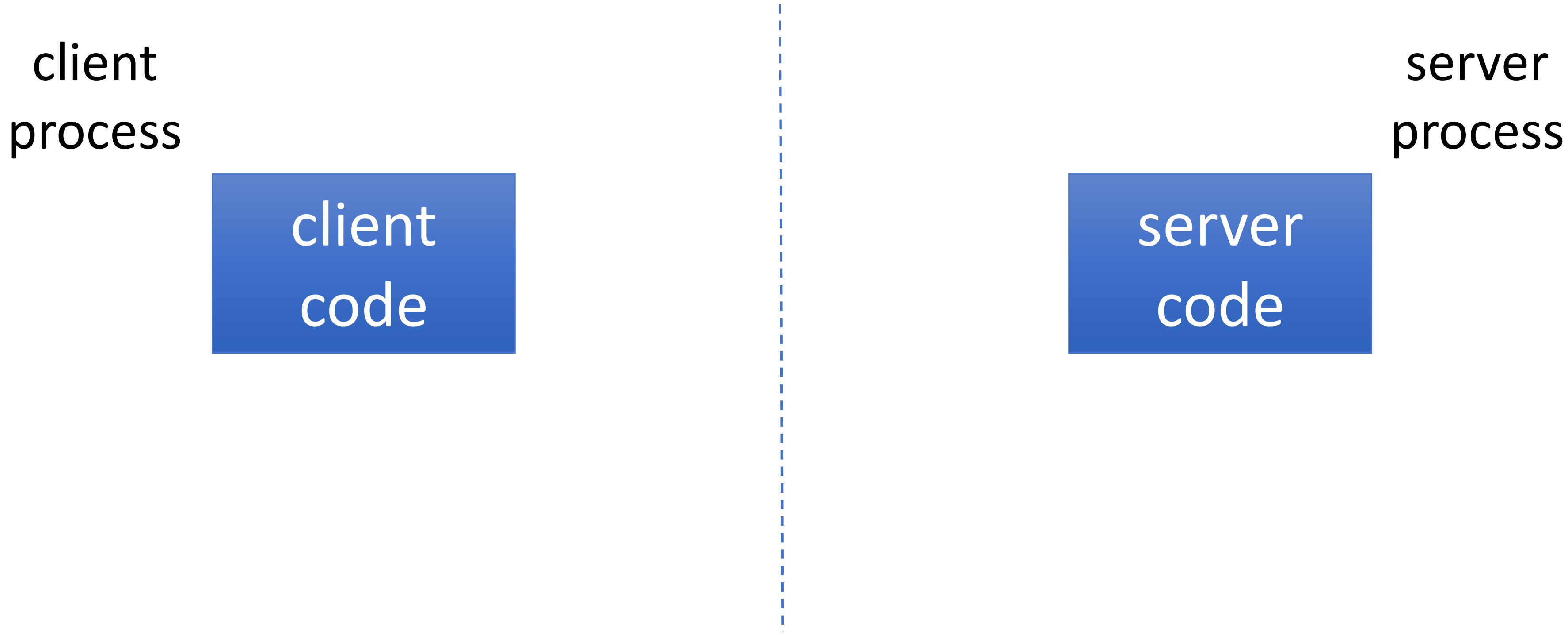
Client Stub

- Sends arguments in call message
- Receives return values in return message

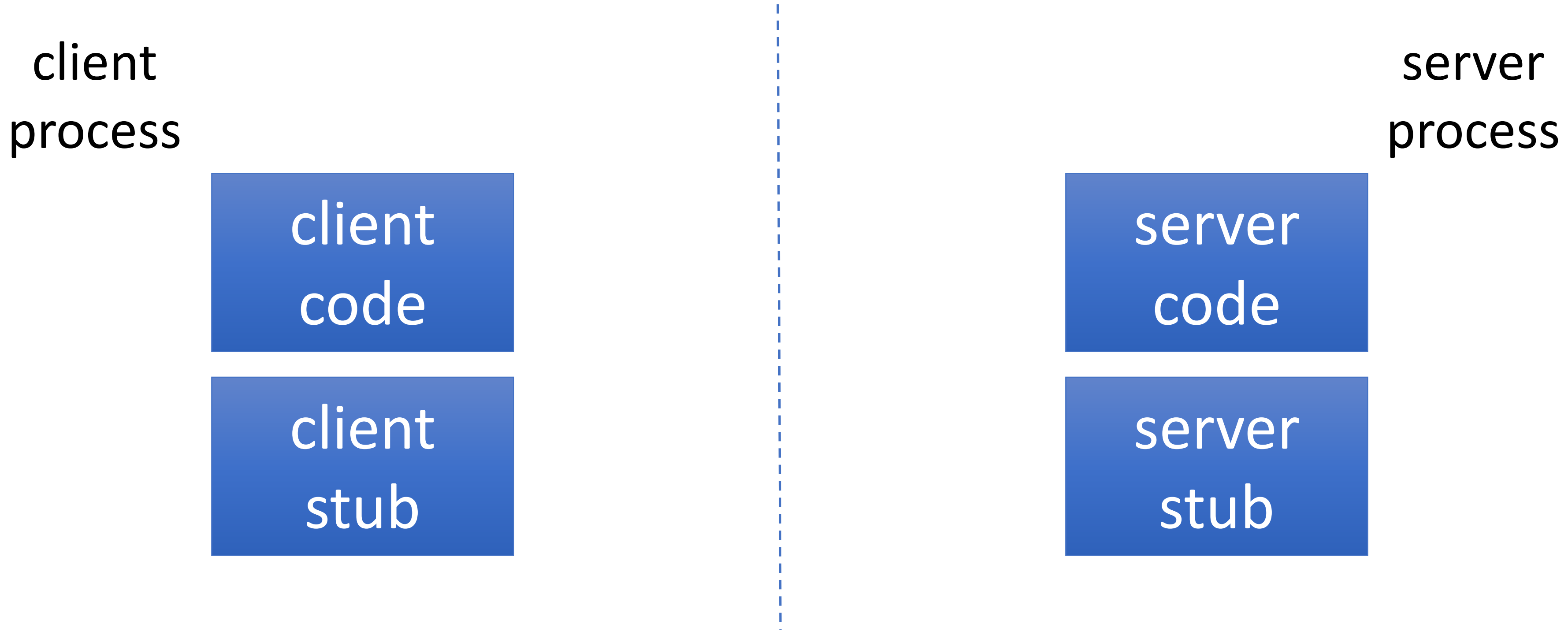
Server Stub

- Receives arguments in call message
- Invokes procedure
- Sends return values in return message

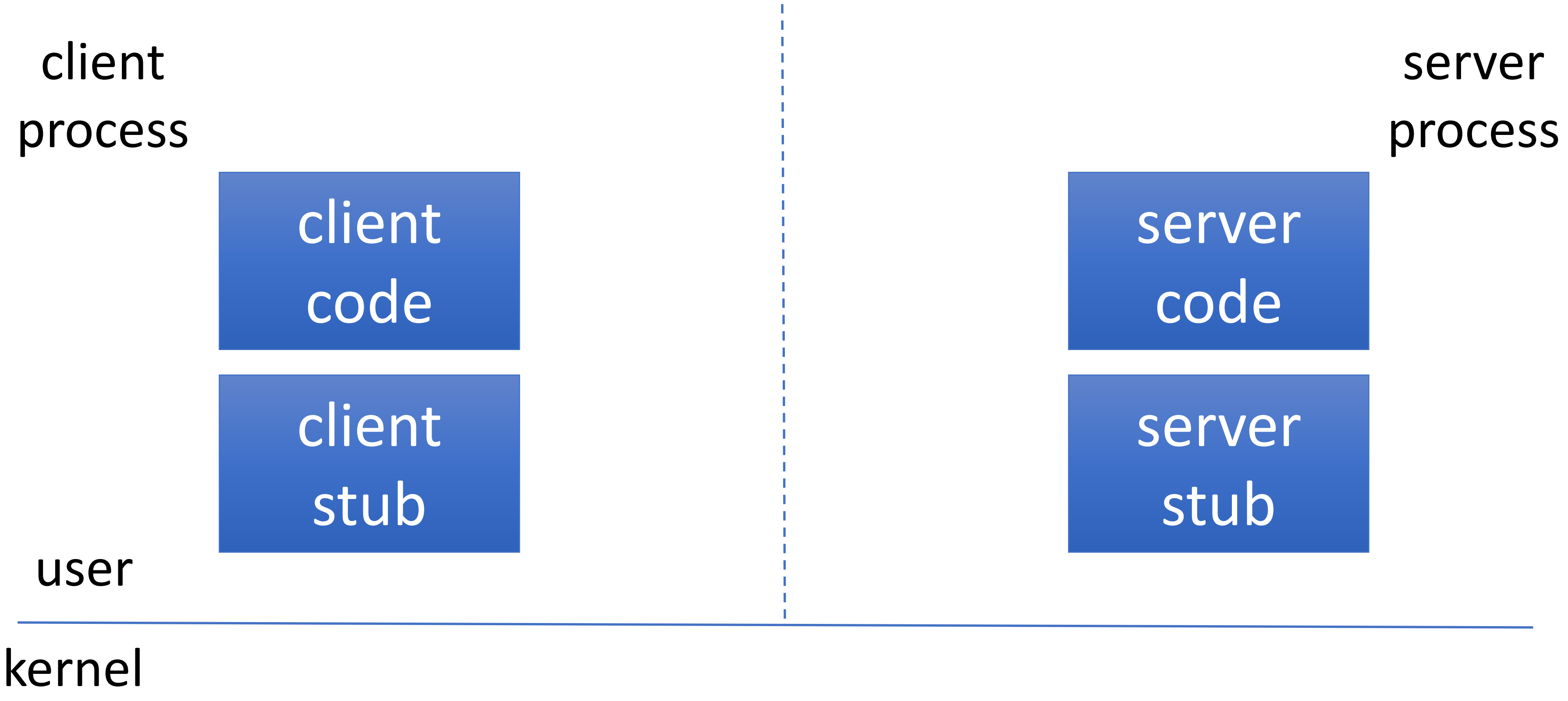
RPC Implementation



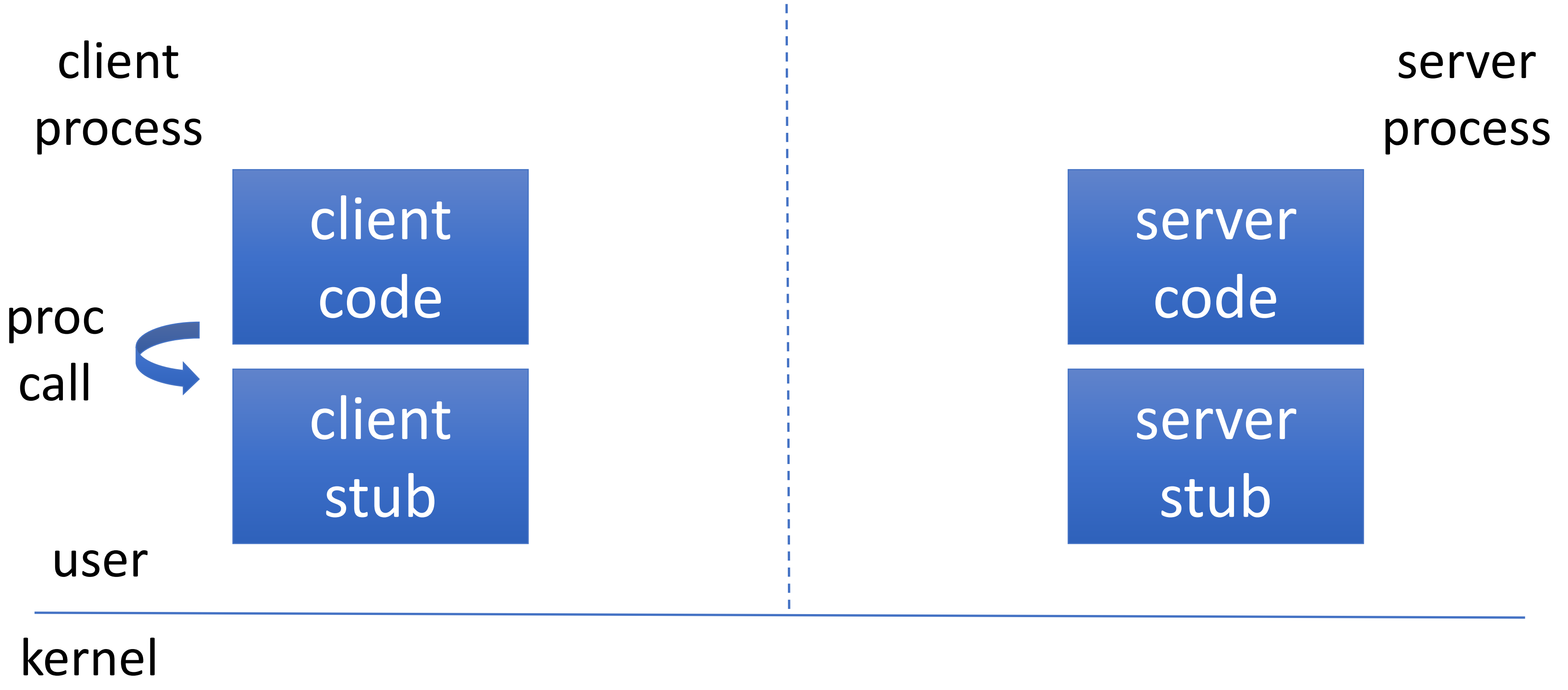
Client and Server Stubs



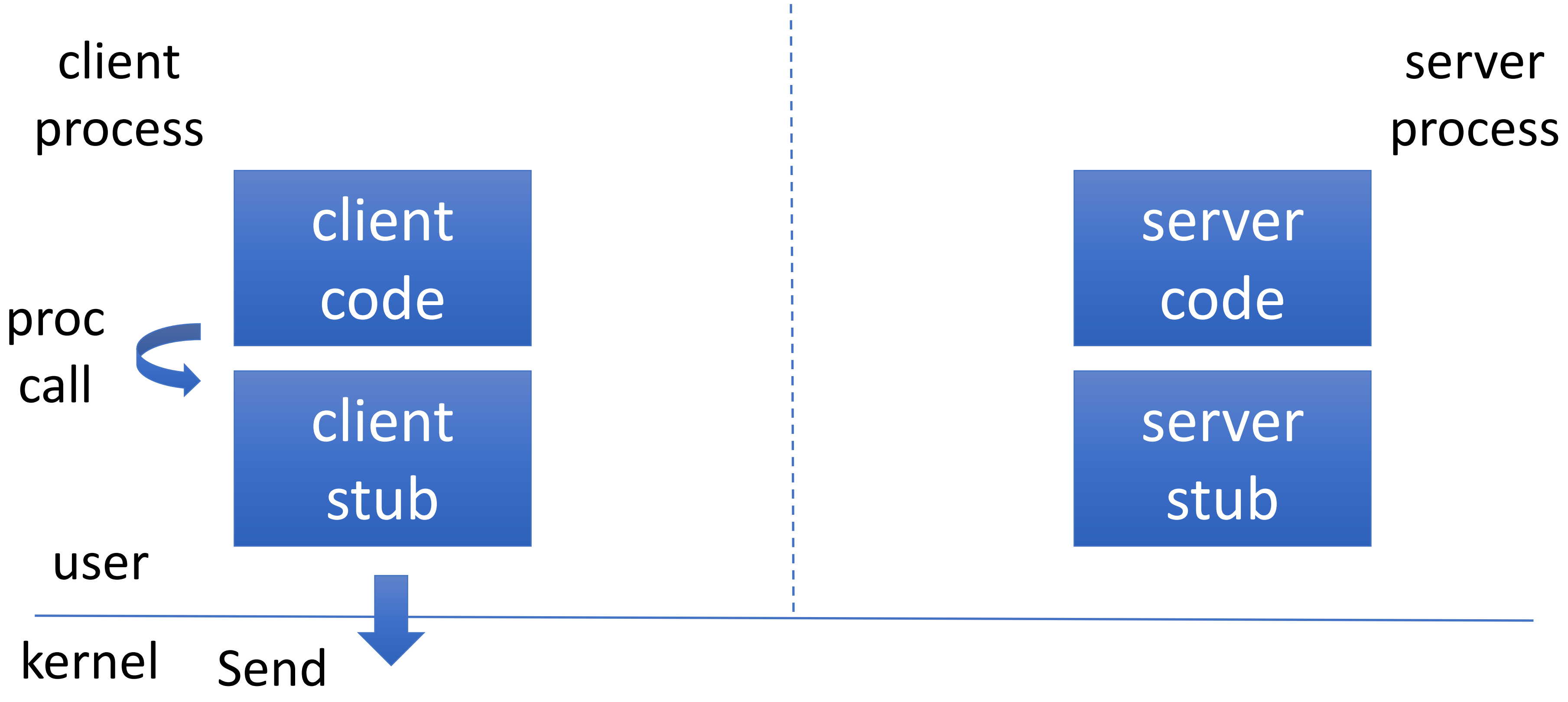
RPC Implementation: Call



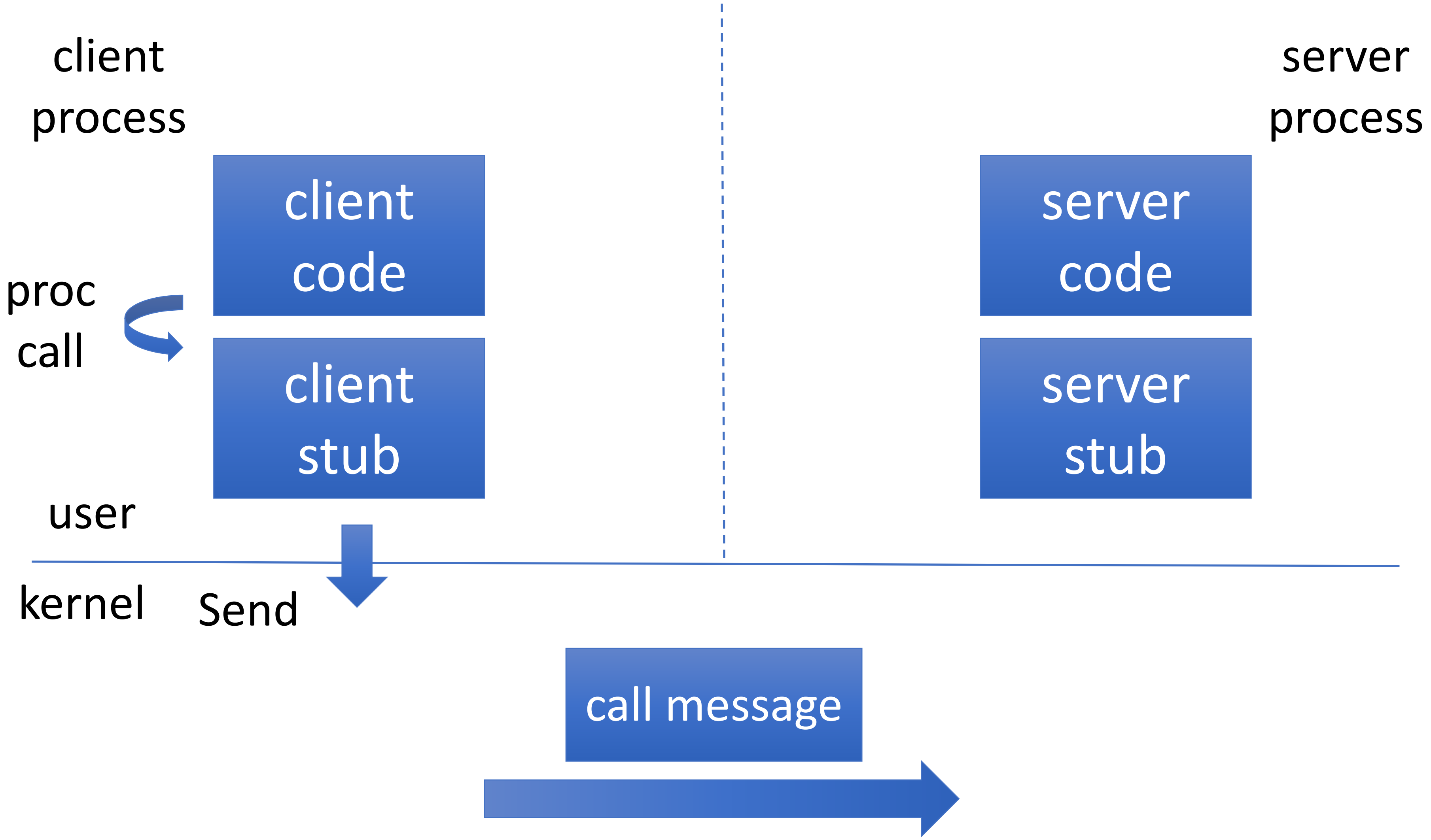
RPC Implementation: Call



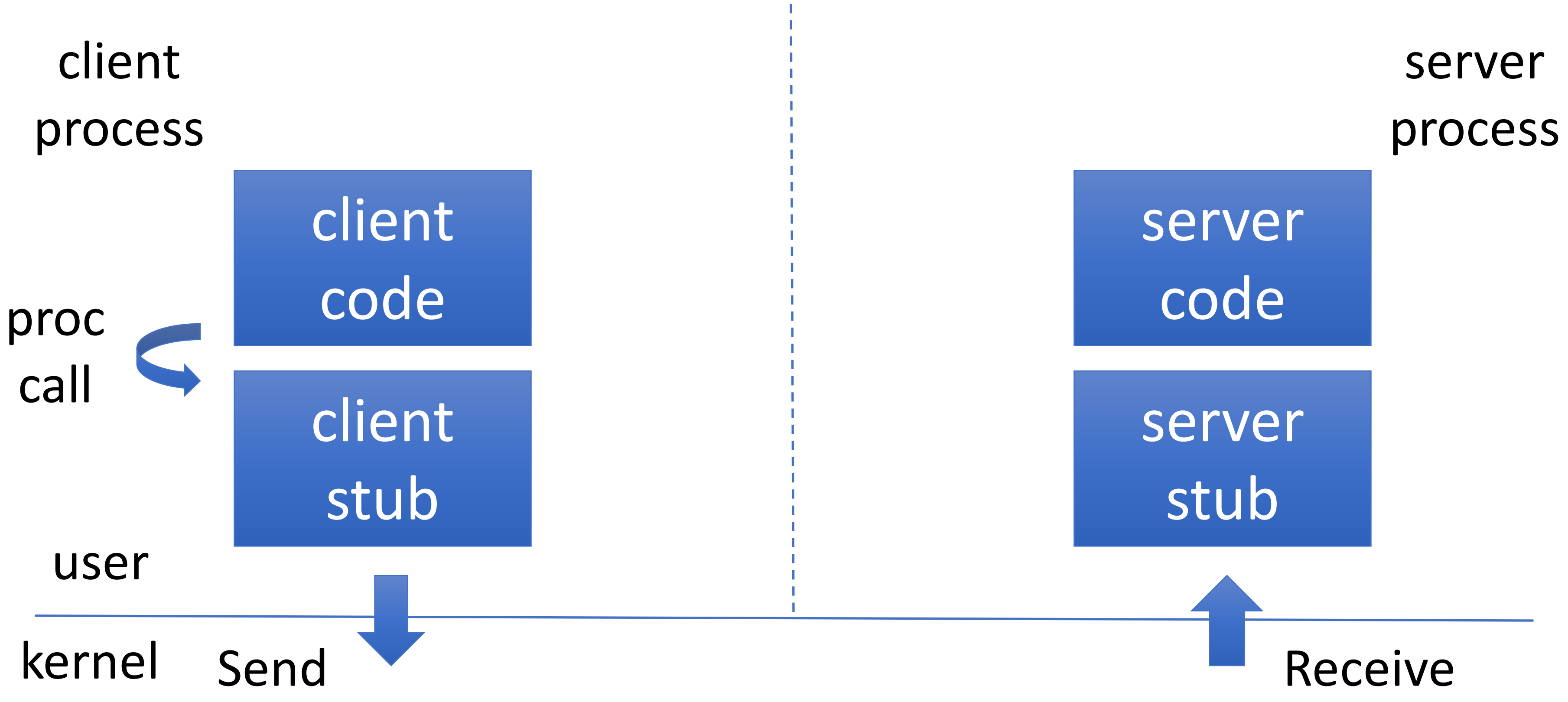
RPC Implementation: Call



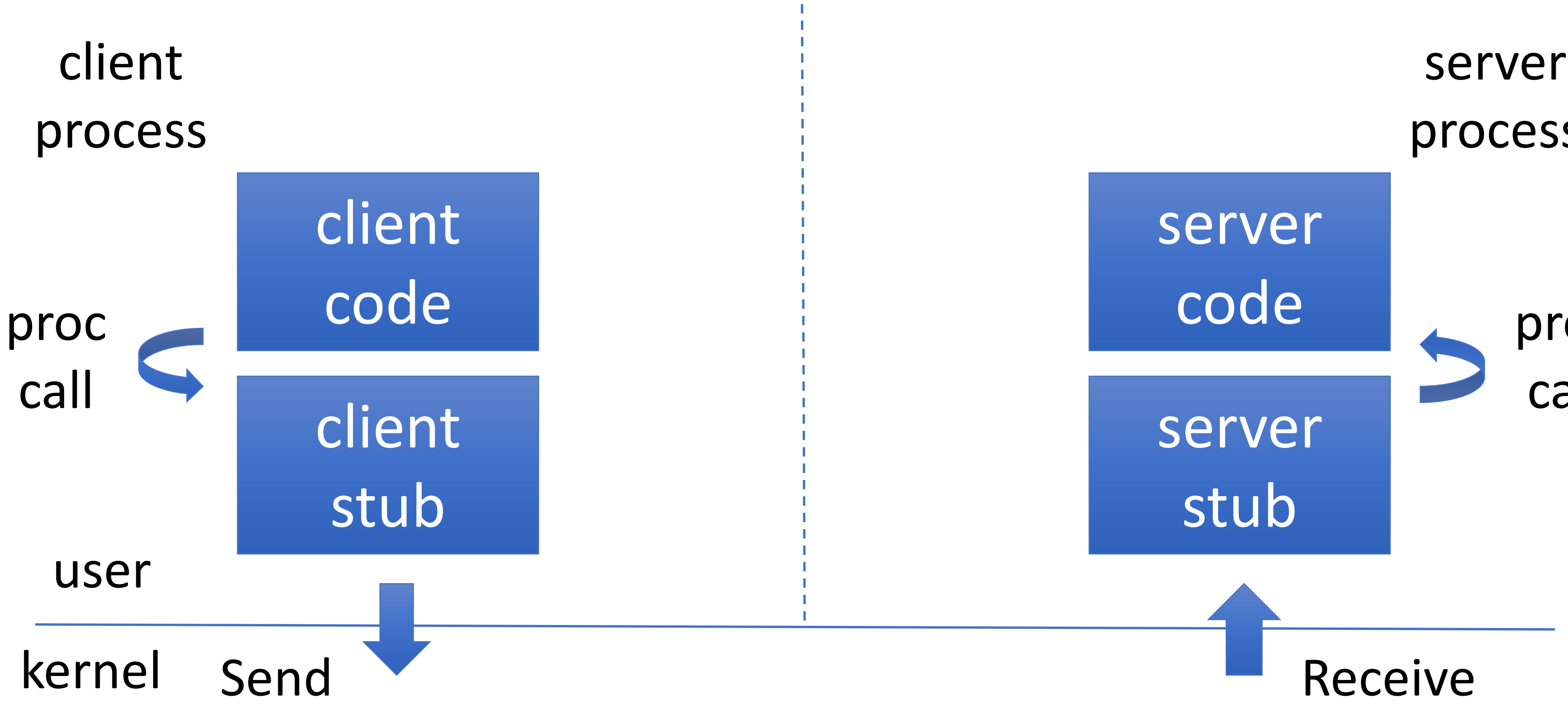
RPC Implementation: Call



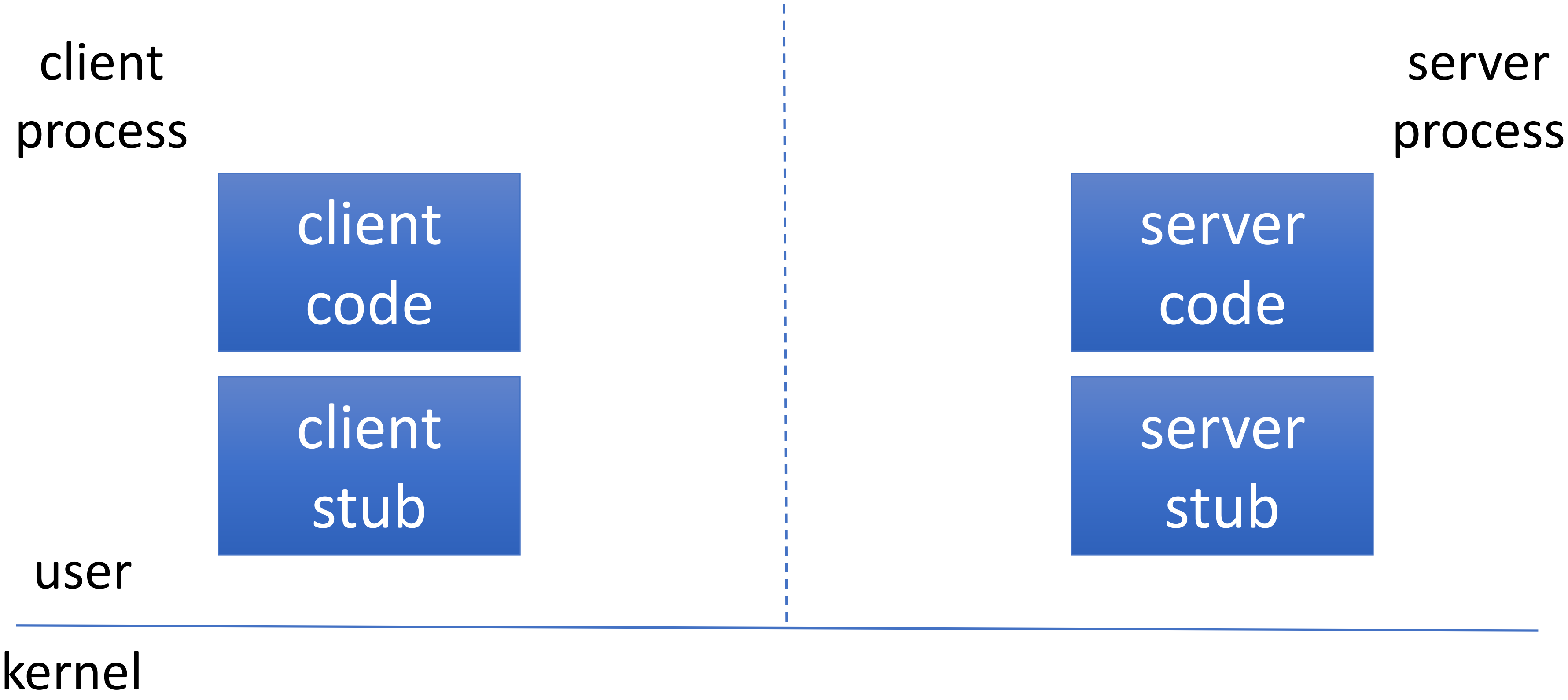
RPC Implementation: Call



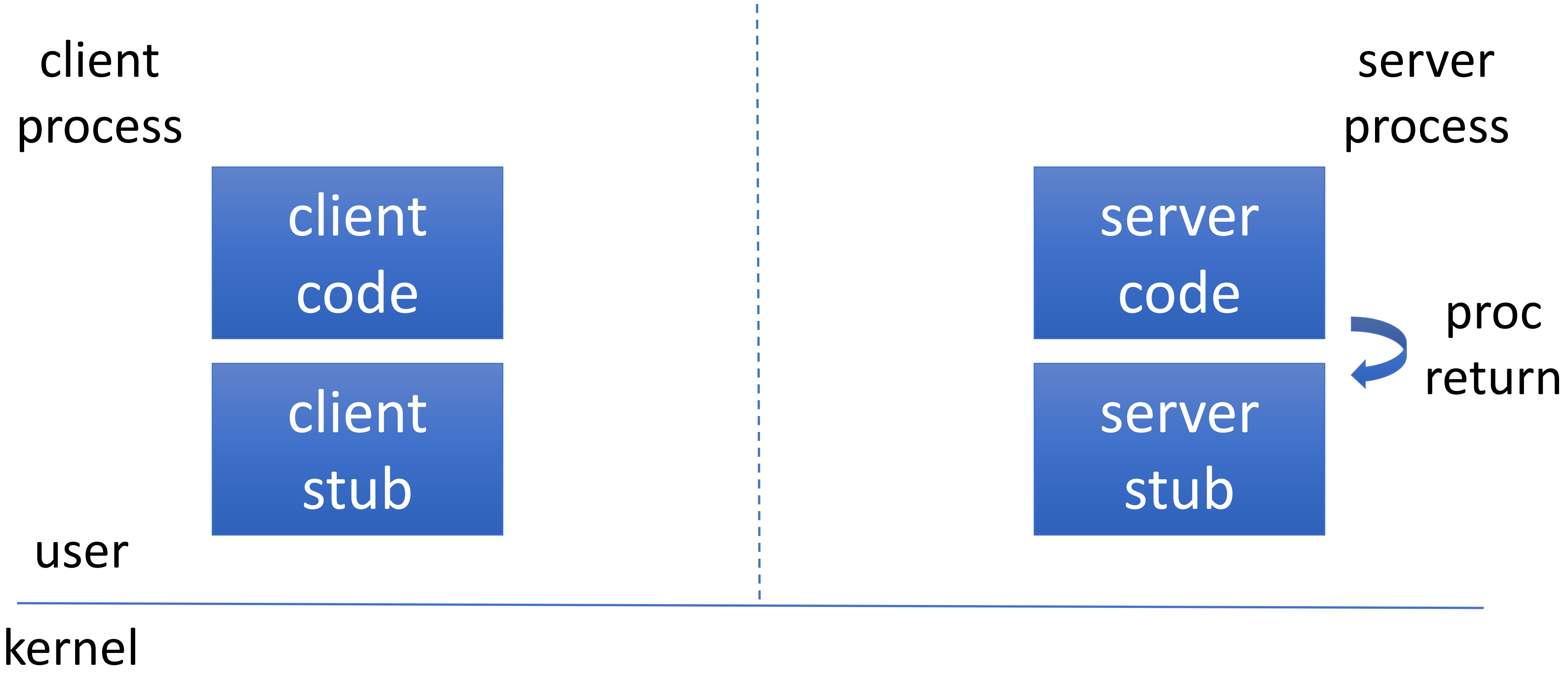
RPC Implementation: Call



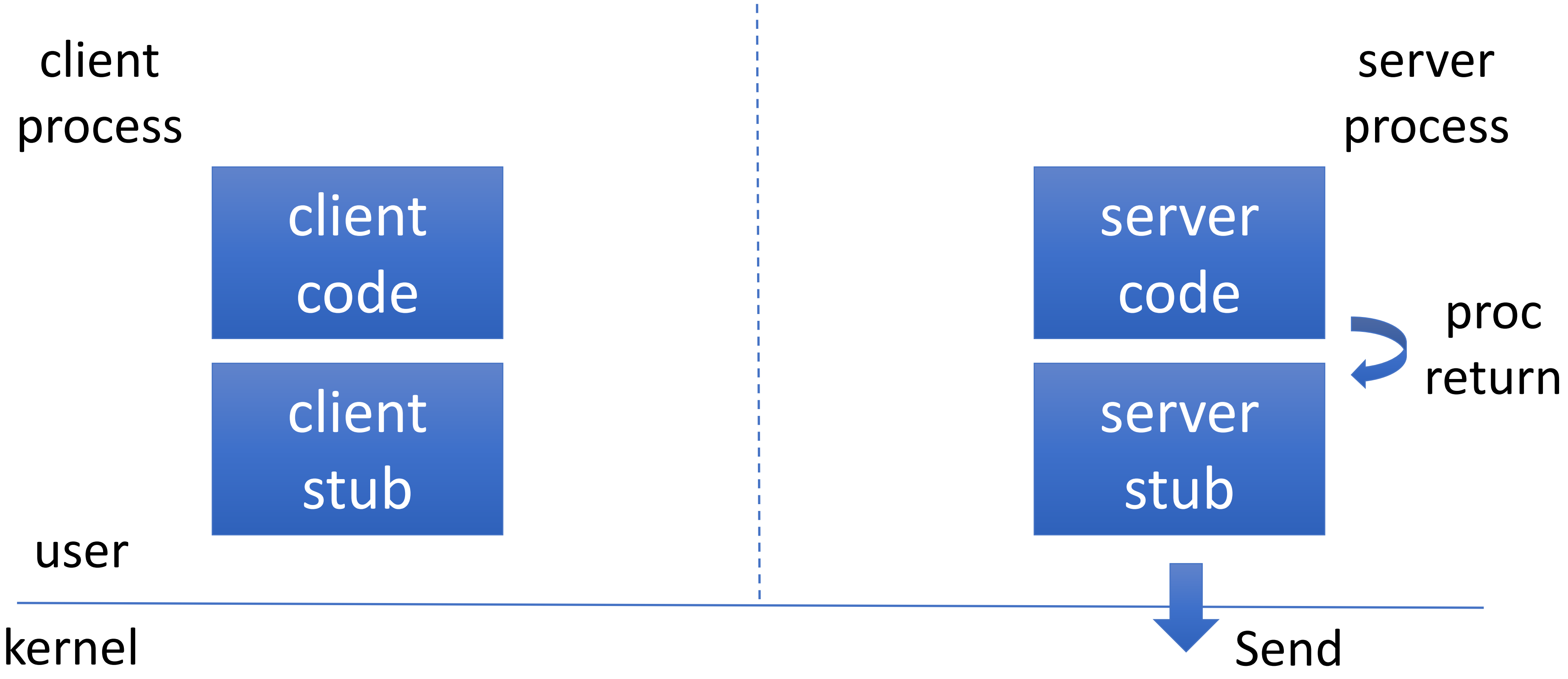
RPC Implementation: Return



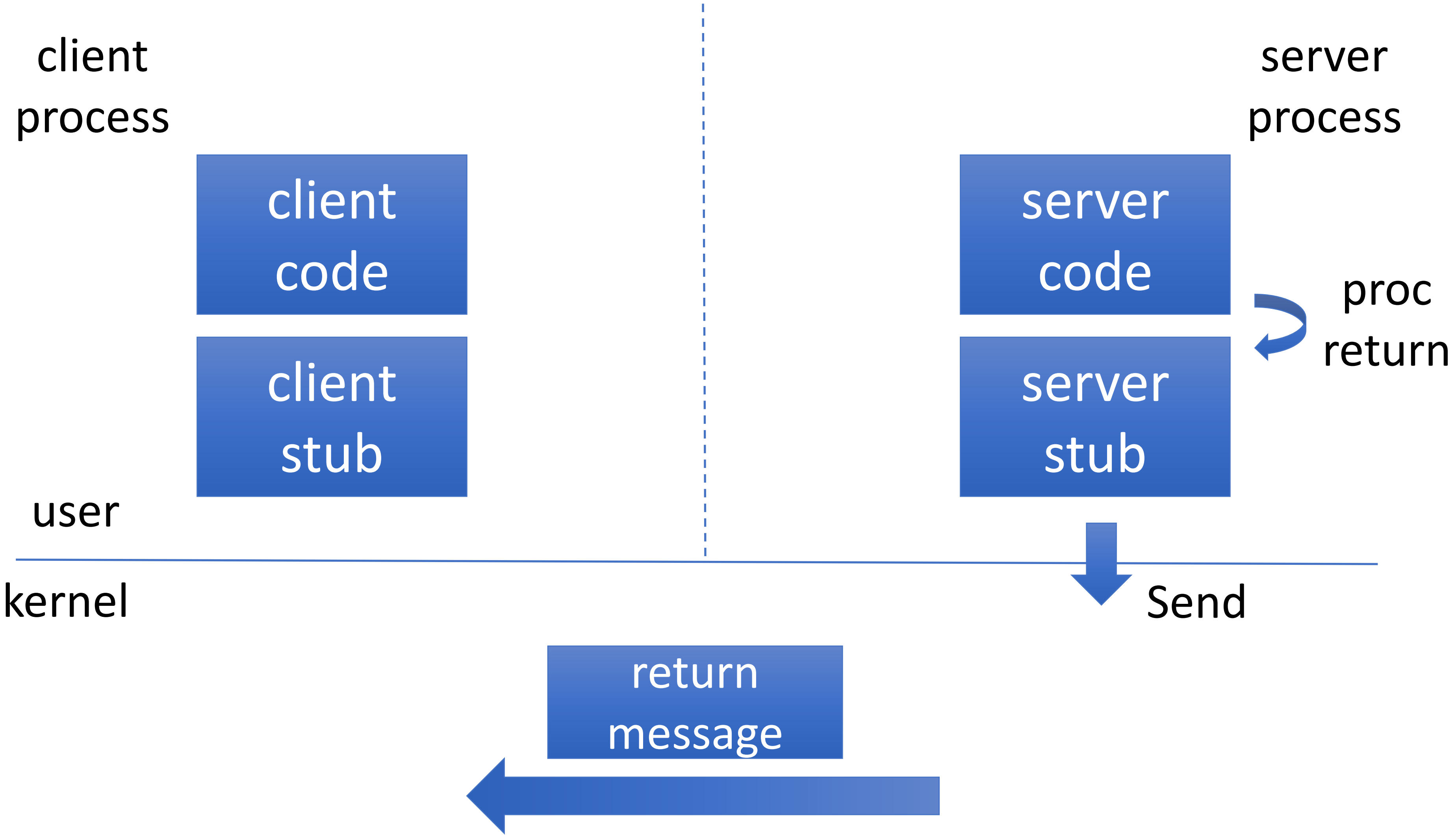
RPC Implementation: Return



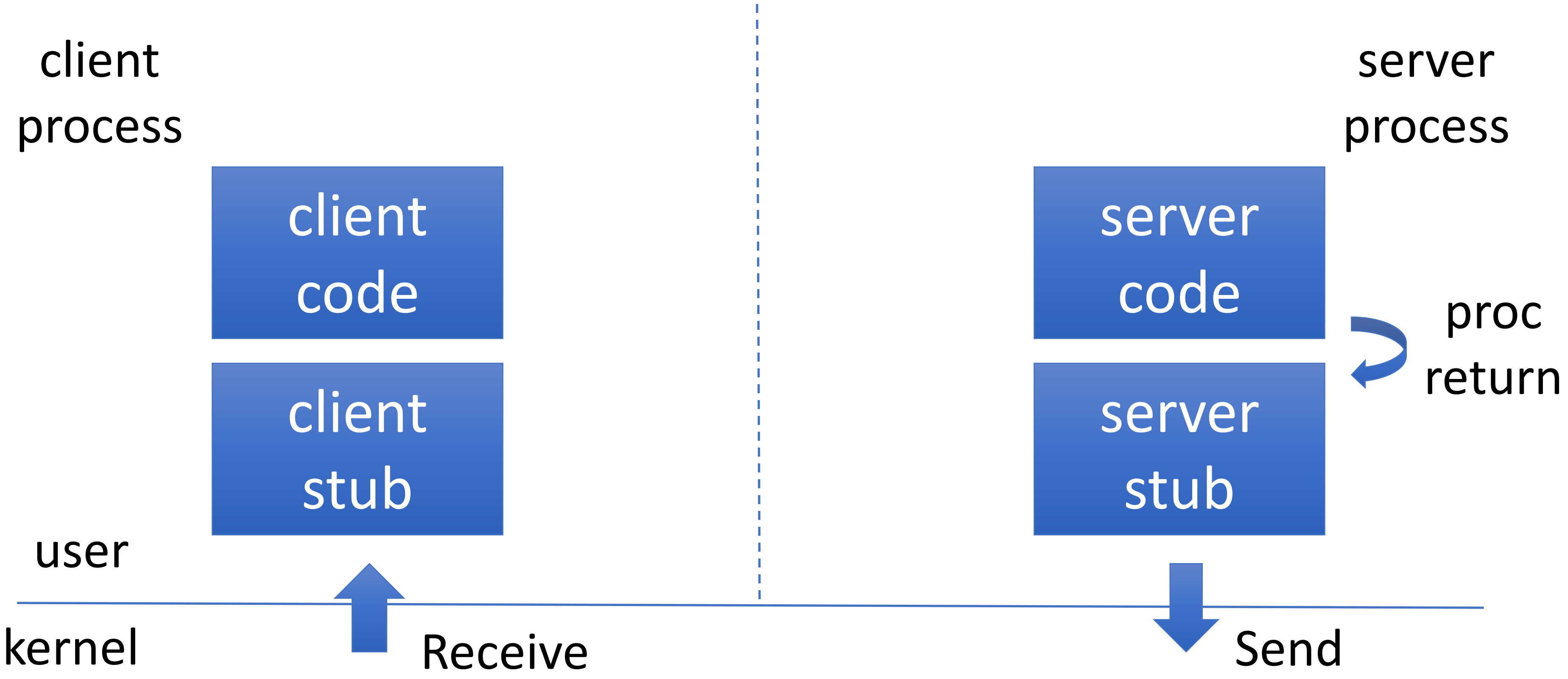
RPC Implementation: Return



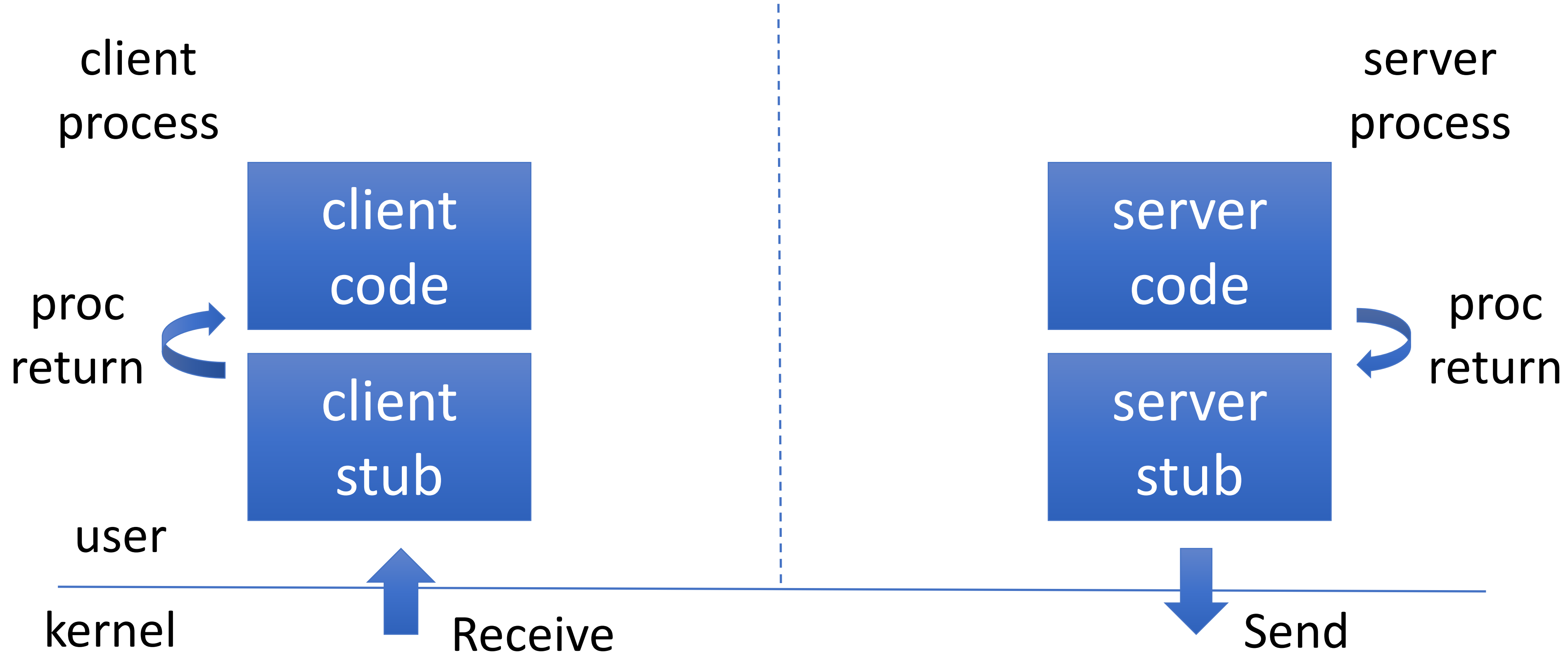
RPC Implementation: Return



RPC Implementation: Return



RPC Implementation: Return



Example 1: Timeserver

- Supports GetTime() and SetTime()

Interface

```
long GetTime()
boolean SetTime(long time)
```

Server Code

```
GetTime()
  {
    return(ReadHardwareClock())
  }
SetTime(time)
  {
    WriteHardwareClock(time)
    return(1)
  }
```

Client Code

```
main()
{
    time = GetTime()
    SetTime(time + 100)
}
```

Message Format

- We already saw:
  - Call message contains arguments
- Must also include which procedure is called

Message Format

Call Message

procno
arg0

Return Message

retval0

Client Stub

```
GetTime()
{
    msg->procno = 1
    Send(msg)
    Receive(msg)
    return(msg->retval0)
}
SetTime(long time)
{
    msg->procno = 2
    msg->arg0 = time
    Send(msg)
    Receive(msg)
    return(msg->retval0)
}
```

Server Stub

```
while(true) do
  {
    Receive(msg)
    switch msg->procno {
      case 1: { time = GetTime()
                msg->retval0 = time
                Send(msg) }

      case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

client code

```
main()
{
  time = GetTime()
  SetTime(time + 100)
}
```

client stub

```
GetTime()
{
  msg->procno = 1
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}

SetTime(long time)
{
  msg->procno = 2
  msg->arg0 = time
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}
```

server code

```
GetTime()
{
  return(ReadHardwareClock())
}
SetTime(time)
{
  WriteHardwareClock(time)
  return(1)
}
```

server stub

```
while(true) do
{
  Receive( msg )
  switch msg->procno {
    case 1: { time = GetTime()
              msg->retval0 = time
              Send(msg) }
    case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

client code

```
main()
{
  time = GetTime()
  SetTime(time + 100)
}
```

client stub

```
GetTime()
{
  msg->procno = 1
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}

SetTime(long time)
{
  msg->procno = 2
  msg->arg0 = time
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}
```

server code

```
GetTime()
{
  return(ReadHardwareClock())
}
SetTime(time)
{
  WriteHardwareClock(time)
  return(1)
}
```

server stub

```
while(true) do
{
  Receive( msg )
  switch msg->procno {
    case 1: { time = GetTime()
              msg->retval0 = time
              Send(msg) }
    case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

client code

```
main()
{
  time = GetTime()
  SetTime(time + 100)
}
```

client stub

```
GetTime()
{
  msg->procno = 1
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}

SetTime(long time)
{
  msg->procno = 2
  msg->arg0 = time
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}
```

server code

```
GetTime()
{
  return(ReadHardwareClock())
}
SetTime(time)
{
  WriteHardwareClock(time)
  return(1)
}
```

server stub

```
while(true) do
{
  Receive( msg )
  switch msg->procno {
    case 1: { time = GetTime()
              msg->retval0 = time
              Send(msg) }
    case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

client code

```
main()
{
  time = GetTime()
  SetTime(time + 100)
}
```

client stub

```
GetTime()
{
  msg->procno = 1
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}

SetTime(long time)
{
  msg->procno = 2
  msg->arg0 = time
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}
```

server code

```
GetTime()
{
  return(ReadHardwareClock())
}
SetTime(time)
{
  WriteHardwareClock(time)
  return(1)
}
```

server stub

```
while(true) do
{
  Receive( msg )
  switch msg->procno {
    case 1: { time = GetTime()
              msg->retval0 = time
              Send(msg) }
    case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

client code

```
main()
{
  time = GetTime()
  SetTime(time + 100)
}
```

client stub

```
GetTime()
{
  msg->procno = 1
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}

SetTime(long time)
{
  msg->procno = 2
  msg->arg0 = time
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}
```

server code

```
GetTime()
{
  return(ReadHardwareClock())
}
SetTime(time)
{
  WriteHardwareClock(time)
  return(1)
}
```

server stub

```
while(true) do
{
  Receive( msg )
  switch msg->procno {
    case 1: { time = GetTime()
              msg->retval0 = time
              Send(msg) }
    case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

client code

```
main()
{
  time = GetTime()
  SetTime(time + 100)
}
```

client stub

```
GetTime()
{
  msg->procno = 1
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}

SetTime(long time)
{
  msg->procno = 2
  msg->arg0 = time
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}
```

server code

```
GetTime()
{
  return(ReadHardwareClock())
}
SetTime(time)
{
  WriteHardwareClock(time)
  return(1)
}
```

server stub

```
while(true) do
{
  Receive( msg )
  switch msg->procno {
    case 1: { time = GetTime()
              msg->retval0 = time
              Send(msg) }
    case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

client code

```
main()
{
  time = GetTime()
  SetTime(time + 100)
}
```

client stub

```
GetTime()
{
  msg->procno = 1
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}

SetTime(long time)
{
  msg->procno = 2
  msg->arg0 = time
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}
```

server code

```
GetTime()
{
  return(ReadHardwareClock())
}
SetTime(time)
{
  WriteHardwareClock(time)
  return(1)
}
```

server stub

```
while(true) do
{
  Receive( msg )
  switch msg->procno {
    case 1: { time = GetTime()
              msg->retval0 = time
              Send(msg) }
    case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

client code

```
main()
{
  time = GetTime()
  SetTime(time + 100)
}
```

client stub

```
GetTime()
{
  msg->procno = 1
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}

SetTime(long time)
{
  msg->procno = 2
  msg->arg0 = time
  Send(msg)
  Receive(msg)
  return(msg->retval0)
}
```

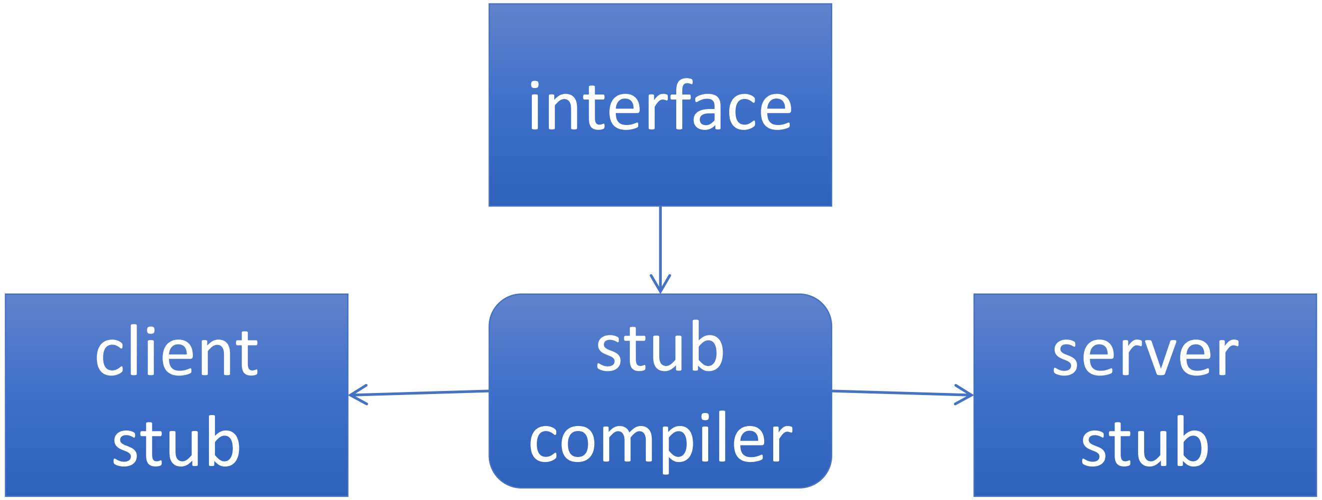
server code

```
GetTime()
{
  return(ReadHardwareClock())
}
SetTime(time)
{
  WriteHardwareClock(time)
  return(1)
}
```

server stub

```
while(true) do
{
  Receive( msg )
  switch msg->procno {
    case 1: { time = GetTime()
              msg->retval0 = time
              Send(msg) }
    case 2: { ret = SetTime(msg->arg0)
               msg->retval0 = ret
               Send(msg) }
  }
```

Note: Stubs Generated Automatically



Let's see how this works in Linux

RPC library

- Make procedure calls on other machines across the network
  - Can also be used for inter-process communication on localhost
  - https://man7.org/linux/man-pages/man3/rpc.3.html

- Beyond the scope of the class to be fluent in RPC. We will just see a simple example.
  - Can continue exploring this in Networking classes (e.g., Distributed systems COMP512, Computer Networks COMP535).

Automatic stub generation

- **rpcbind** — universal addresses to RPC program number mapper
  - A server that converts RPC program numbers into universal addresses.
  - It must be running on the host to be able to make RPC calls on a server on that machine.
  - **sudo apt-get install rpcbind**

Automatic stub generation cont’d

rpcgen – tool that generates remote program interface modules.

- compiles source code written in the RPC language.
- RPC language is similar in syntax and structure to C.
- produces one or more C language source modules, which are then compiled by a C compiler.
- ☺ rpcgen reduces the development time that would otherwise be spent developing low-level routines.
  - Have a look at a tutorial

Another example: Sum Server

- Client
    - sends 2 integers: a and b
- Server
    - replies with a+b

Step 1: Defining the data structures with XDR routines, in add.x file

```
//add.x

struct numbers{
  int a;
  int b;
};


program ADD_PROG{                    /* program number */
  version ADD_VERS{                  /* version number */
    int add(numbers)=1;              /* procedure */
  }=1;
}=0x12345;
```

Step 2: Compile XDR routines into C code

```
//add.x

struct numbers{
  int a;
  int b;
};


program ADD_PROG{                      /* program number */
  version ADD_VERS{                    /* version number */
    int add(numbers)=1;                /* procedure */
  }=1;
}=0x12345;
```

```
~/COMP310-2023/RPC$ rpcgen -a -C add.x
```

Step 3: Modify the server/client code.

```
balmau@discslab-server1:~/COMP310-2023/RPC/RPC-class$ ls
add_client.c  add.h      add_svc.c  add_xdr.c
add_clnt.c    add_server.c  add.x      Makefile.add
```

- add_client.c : client code that we need to implement
- add_clnt.c : automatically generated stub code – No need to change
- add_server.c : server code that we need to implement
- add_svc.c : automatically generated stub code – No need to change

Step 3.1: Modify the server code add_server.c

```c
#include "add.h"


int * add_1_svc(numbers *argp, struct svc_req *rqstp)
{
        static int  result;
        /*
         * insert server code here
         */
          return &result;
}
```

Step 3.1: Modify the server code add_server.c

```c
#include "add.h"


int * add_1_svc(numbers *argp, struct svc_req *rqstp)
{
        static int  result;
        /*
         * insert server code here
         */
        printf("add(%d,%d) is called\n", argp->a, argp->b);
        result = argp->a + argp->b;
        return &result;
}
```

Step 3.2: Modify the client code add_client.c

```c
int main (int argc, char *argv[]) {
  char *host;
  if (argc < 2) {
    printf ("usage: %s server_host\n",
argv[0]);
    exit (1);
  }
  host = argv[1];
  add_prog_1 (host);
  exit (0);
}
```

```c
void add_prog_1(char *host) {
  CLIENT *clnt;
  int  *result_1;
  numbers  add_1_arg;

#ifndef DEBUG
  clnt = clnt_create (host, ADD_PROG,
ADD_VERS, "udp");
  if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
  }
#endif  /* DEBUG */

  result_1 = add_1(&add_1_arg, clnt);
  if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
  }
#ifndef DEBUG
  clnt_destroy (clnt);
#endif   /* DEBUG */
}
```

Step 3.2: Modify the client code add_client.c

```
int main (int argc, char *argv[]) {
  char *host;
  if (argc < 4) {
    printf ("usage: %s server_host\n", 
argv[0]);
    exit (1);
  }
  host = argv[1];
  add_prog_1 (host);
  add_prog_1 (host, atoi(argv[2]), 
atoi(argv[3]));

  exit (0);
}
```

```
void add_prog_1(char *host, int x, int y) {
  CLIENT *clnt;
  int  *result_1;
  numbers  add_1_arg;

//...

  add_1_arg.a=x;
  add_1_arg.b=y;

  result_1 = add_1(&add_1_arg, clnt);
  if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
  }  else {
    printf("Result:%d\n", *result_1);
  }
//...}
```

Step 4: Compile **C code**

```
balmau@discslab-server1:~/COMP310-2023/RPC/RPC-class$ ls
add_client.c  add.h      add_svc.c  add_xdr.c
add_clnt.c    add_server.c  add.x      Makefile.add
```

```
~/COMP310-2023/RPC$ make -f Makefile.add
```

Step 5: Launch the server

```
balmau@discslab-server1:~/COMP310-2023/RPC$ ./add_server
```

Step 6: Use the client

Numbers we want to add

Open new terminal window

```
balmau@discslab-server1:~/COMP310-2023/RPC$ ./add_client localhost 1 2
Result:3
```

Server process is on the same machine, so we use localhost

Meanwhile, in the server window:

```
balmau@discslab-server1:~/COMP310-2023/RPC$ ./add_server
add(1,2) is called
```

Optional Homework

- Use the same process to create the Timeserver in example 1
- *Hint: The linked rpcgen tutorial will help*

Key Concepts for Today

- Interprocess communication
- Message passing
- Remote procedure call

Further Optional Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 25 – 32 (inclusive) https://pages.cs.wisc.edu/~remzi/OSTEP/

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Maurice Herlihy (Brown University)