Problems with Send and Receive

- Low level
  - programmer is engaged in I/O
  - server often not modular
  - takes 2 calls to get what you want (send, followed by receive) -- error prone
- Solution
  - use procedure calls -- familiar model

Remote Procedure Call (RPC)

- Allow procedure calls to other machines
  - servicing of procedure remote
  - caller blocks until procedure finished, as usual
  - simpler than explicit message passing
- Complications
  - caller and receiver in different address spaces
  - parameter passing
  - where is the server?
  - what about crashes?

Recall: Programming Client/Server Applications (General Outline)

Outline of Client code

```
while (1)   {
  build request
  send(request, server)
  receive(reply)
  do something
}
```

Outline of Server code

```
while (1)  {
  receive(request)
  switch(request.type)
  case FOO:
     …
     send(client, reply1)
  case BAR:
     …
     send(client, reply2)
  etc.
```

Programming Client/Server Applications with RPC

Outline of Client code

```
while (1)   {
  reply = foo(params)
  do something
}
```

Note: Server is written as collection of several procedures

Outline of Server code

```
foo(params) {
    ….
    return reply1;
}
bar (params) {
    ….
    return reply2;
}
```

Basics of RPC Implementation

- Goal: provide complete transparency to RPC user
  - Implementation replaces a normal procedure call with:
    - pack arguments (including function) into a message via a "stub function"
      - may need to worry about byte ordering, linked lists, etc
    - send message to server; block waiting for reply
      - implemented via explicit message passing (send/receive)

Basics of RPC Implementation

- Goal: provide complete transparency
  - On receipt at server: unpack and push parameters onto the stack, call function **(create new thread)**
    - Implemented by creating a thread that calls a stub function
      - picture following does not show stub function on server side, for simplicity
  - Server then sends reply to client with results of function
  - On receipt of reply at client: put result where it belongs, unblock client

RPC Implementation

```
Client
while (1)   {
  reply = foo(params)
  do something
}
```

```
Server
foo(params) {
    ….
    return reply1
}
```

RPC Implementation

```
Client
while (1)   {
  reply = foo(params)
  foo_stub(params, &reply)
  do something
}
foo_stub(params, &reply) {
  msg.func = foo
  msg.data[0] = param1
  msg.data[1] = param2
  send(Server, msg)
  receive(Server, result)
  reply = result.returnVal
}
```

```
Server
foo(params) {
   ….
   return reply1
}
```

RPC Implementation

```
Client
while (1)   {
  reply = foo(params)
  foo_stub(params, &reply)
  do something
}
foo_stub(params, &reply) {
  msg.func = FOO
  msg.data[0] = param1
  msg.data[1] = param2
  send(Server, msg)
  receive(Server, result)
  reply = result.returnVal
}
```

```
Server
foo(params, &returnVal) {
   ….
   return reply1
   returnVal = reply1
}
RPC_server( ) {
  receive((Client = ANY_SOURCE), msg)
  params = msg.params
  switch(msg.func) {
     case FOO:
        t = thread_create(foo, params, &retVal)
        thread_join(t)
        msg.returnVal = retVal
        send(Client, msg)
   }
}
```

RPC Implementation Issues

- Weakly typed languages
  - E.g., C --- what to do if unbounded array passed to RPC?
  - Pointers across different machines?
- Communication via global variables impossible
- Binding
  - How does client know where server is?
    - One solution: use a database
- Failures?
  - What if function is partially executed, or executed twice, or executed never?

RPC Parameter Passing

- Client machine may be a different architecture than server
  - we will ignore this issue – one side must convert data if byte ordering is an issue
- Parameter issues
  - what parameter passing style should be provided?
  - can be important performance issue
  - not as easy as it seems at first glance

Call by Value

- Simple semantics
- Just package up the args, and send them
  - can be problematic (efficiency-wise) if pointer parameter points to a complex data type, e.g., graph or list
- Server uses these args
  - doesn't need to send them back

Call by Reference

- What do pointers mean across machines?
  - remember, they mean nothing across address spaces
- Could send back message to client on each reference
    - SLOW!
    - Never used for RPC

Call by Copy/Restore

- Similar to call by reference
  - parameter copied in, same as call by value
    - same disadvantages of having to copy entire structures
  - but when procedure finished, copy parameter back to caller
  - not quite same as call by reference
  - method of choice for “reference parameters” when using RPC

(Contrived) example of how call by reference and call by copy-restore can differ

```
int a;
foo(int x) {
  x = 2; a = 0;
}
int main( ) {
  a = 1; foo(a); print(a)
}
```

Call by reference outputs 0; call by copy-restore outputs 2

Failures

- Many things can go wrong with RPC, e.g., server crash
  - How do we know, from client's perspective, if the server crashed?
  - Supposing we know the server crashed, what do we do from the client side?
    - Run RPC again?
    - Something else?

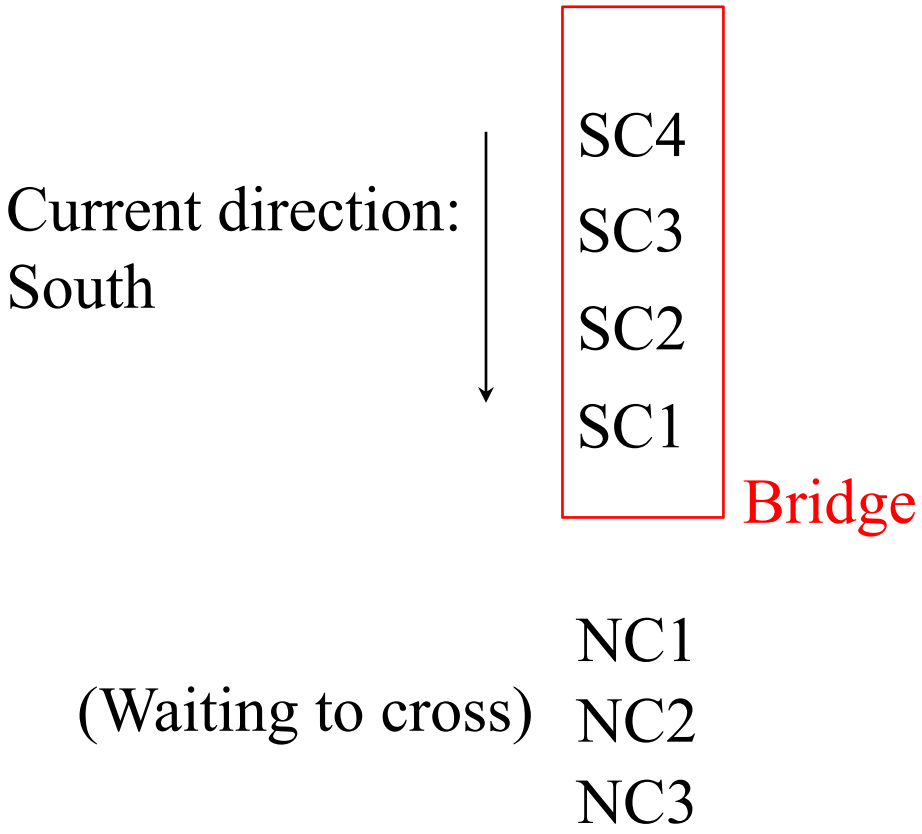
Rendezvous

- Similar to RPC
  - Key difference: no new process created on the server (*but unlike RPC, there is synchronization*)
  - Caller side is the same as with RPC
  - Server looks roughly as follows:

```
in op1(…)
    execute code for op1
[] op2(…)
    execute code for op2
ni
```

  - Server blocks until >= 1 pending invocation on any op (can be implemented via UNIX *select*)

One Lane Bridge Problem: Picture



One Lane Bridge with Monitors

```
monitor Bridge
    void Arrive(int)
    void Exit(int)
    int numCars = 0, int currentDirection = 0, condition headOn

    void Arrive(int direction)
       while (currentDirection != direction and numCars > 0)
         Wait(headOn)
       if (numCars == 0)
         currentDirection = direction
       numCars ++

    void Exit(int direction)
        numCars –
        Broadcast(headOn)
```

Cars invoke Bridge.Arrive(direction) and Bridge.Exit(direction)

One Lane Bridge with RPC

```
module Cars
  call Bridge.Arrive(direction) or call Bridge.Exit(direction)  // RPCs on Client

monitor Bridge  // Executes on server
    void Arrive(int)
    void Exit(int)
    int numCars = 0, currentDirection = 0, cond headOn

    void Arrive(int direction)
      while (currentDirection != direction and numCars > 0)
        Wait(headOn)
      if (numCars == 0)
        currentDirection = direction
      numCars ++

    void Exit(int direction)
       numCars –
       Broadcast(headOn)
```

Note: Arrive and Exit
need to be monitor functions

One Lane Bridge with Rendezvous

```
module Cars // Executes on client
  call Bridge.Arrive(direction) or send Bridge.Exit(direction)  // Remote invocations


module Bridge   // Just a class---not a monitor!  Executes on server
    void Arrive(int)
    void Exit(int)
    int numCars = 0, currentDirection = 0

    process ManageBridge  {
      while(true)
         in Arrive(direction) and (currentDirection == direction or numCars == 0)
            if (numCars == 0)
              currentDirection = direction
            numCars++
         [] Exit(direction)
            numCars--
         ni
    }
```