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)

(called RMI in Java)

- 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 machine providing the service?
  - what about crashes?

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
}
```

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**
  - RPC implementation must make a remote call look like a local (regular) procedure call
  - **Implementation** replaces a normal procedure call with:
    - pack arguments (including function to be executed) into a message via a "stub function"
      - may need to worry about dynamic data structures as well as low-level concerns such as byte ordering
    - send message to server; block waiting for reply
      - implemented via explicit message passing (send/receive)

RPC User's View

```
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
}
```

User does not know anything about foo_stub

User does not know anything about foo_stub

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
  - 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)
  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)
  switch(msg.func) {
     case FOO:
        t = thread_create(foo_stub', params)
}
foo_stub'(params) {
   foo(params, returnVal)
   msg.returnVal = returnVal
   send(Client, msg)
}
```

User does not know anything about RPC_server

One case per function

User does not know anything about foo_stub'

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 servicing machine 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?
  - they mean nothing across address spaces on the same machine, let alone on different machines
- Could send back message to client on each reference
  - **Would be 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 the 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( ) {
  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
    - Unlike RPC, built-in synchronization between operations
  - Caller side is the same as with RPC
  - Server design is as follows:

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

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

Bank Account Problem

- Several people (threads) share a savings account
  - Current balance is sum of all deposits to date minus sum of all withdrawals to date
  - Balance must never become negative
  - Deposits don't delay (except for mutual exclusion)
  - Withdrawals must delay until sufficient funds are in the account

Bank Account with Monitors

```
monitor Account
  int balance = INIT_BALANCE
  Cond bank

  Withdraw(amount) {
      while (amount > balance)
          Wait(bank)
      balance = balance – amount
  }

  Deposit(amount) {
      balance = balance + amount
      Broadcast(bank)
  }
end monitor
```

Bank Account with RPC

```
module Customer    // Executes on client
  Deposit(amount) or Withdraw(amount)    // Customer calls one or the other

module Bank    // Executes on server
    monitor Account
      void Deposit(int)
      void Withdraw(int)
      int balance = INIT_BALANCE,  Cond bank

      Withdraw(int amount)
        while (amount > balance)
          Wait(bank)
        balance = balance – amount
      Deposit(int amount)
        balance = balance + amount
        Broadcast(bank)
```

Note: Deposit and Withdraw need to be monitor functions (not shown here; refer to prev. slide)

Bank Account with Rendezvous

```
module Customer    // Executes on client
  Deposit(amount) or Withdraw(amount)    // Remote invocations

module Bank    // Just a class---not a monitor!  Executes on server
    void Deposit(int)
    void Withdraw(int)
    int balance = INIT_BALANCE

    process Teller {
       while (true)
          in Deposit(amount)
              balance = balance + amount
          [] Withdraw(amount) and balance >= amount
              balance = balance – amount
          ni
    }
```