RPC and Rendezvous

Andrews, Chapter 08

```
module TimeServer
  op get_time() returns int;  # retrieve time of day
  op delay(int interval);     # delay interval ticks
body
  int tod = 0;          # the time of day
  sem m = 1;            # mutual exclusion semaphore
  sem d[n] = ([n] 0);   # private delay semaphores
  queue of (int waketime, int process_id) napQ;
  ## when m == 1, tod < waketime for delayed processes

  proc get_time() returns time {
    time = tod;
  }

  proc delay(interval) {    # assume interval > 0
    int waketime = tod + interval;
    P(m);
    insert (waketime, myid) at appropriate place on napQ;
    V(m);
    P(d[myid]);   # wait to be awakened
  }

  process Clock {
    start hardware timer;
    while (true) {
      wait for interrupt, then restart hardware timer;
      tod = tod+1;
      P(m);
      while (tod >= smallest waketime on napQ) {
        remove (waketime, id) from napQ;
        V(d[id]);  # awaken process id
      }
      V(m);
    }
  }
end TimeServer
```

Figure 8.1 A time server module.

```
module FileCache   # located on each diskless workstation
  op read(int count; result char buffer[*]);
  op write(int count; char buffer[]);
body
  cache of file blocks;
  variables to record file descriptor information;
  semaphores for synchronization of cache access (if needed);

  proc read(count,buffer) {
    if (needed data is not in cache) {
      select cache block to use;
      if (need to write out the cache block)
        FileServer.writeblk(...);
      FileServer.readblk(...);
    }
    buffer = appropriate count bytes from cache block;
  }

  proc write(count,buffer) {
    if (appropriate block not in cache) {
      select cache block to use;
      if (need to write out the cache block)
        FileServer.writeblk(...);
    }
    cache block = count bytes from buffer;
  }
end FileCache
```

Figure 8.2 (a) Distributed file system: File cache.

```
module FileServer   # located on a file server
  op readblk(int fileid, offset; result char blk[1024]);
  op writeblk(int fileid, offset; char blk[1024]);
body
  cache of disk blocks;
  queue of pending disk access requests;
  semaphores to synchronize access to the cache and queue;
  # N.B. synchronization code not shown below

  proc readblk(fileid, offset, blk) {
    if (needed block not in the cache) {
      store read request in disk queue;
      wait for read operation to be processed;
    }
    blk = appropriate disk block;
  }

  proc writeblk(fileid, offset, blk) {
    select block from cache;
    if (need to write out the selected block) {
      store write request in disk queue;
      wait for block to be written to disk;
    }
    cache block = blk;
  }

  process DiskDriver {
    while (true) {
      wait for a disk access request;
      start a disk operation; wait for interrupt;
      awaken process waiting for this request to complete;
    }
  }
end FileServer
```

Figure 8.2 (b) Distributed file system: File server.

```
optype stream = (int);  # type of data stream operations

module Merge[i = 1 to n]
  op in1 stream, in2 stream;  # input streams
  op initialize(cap stream);  # link to output stream
body
  int v1, v2;    # input values from streams 1 and 2
  cap stream out;  # capability for output stream
  sem empty1 = 1, full1 = 0, empty2 = 1, full2 = 0;

  proc initialize(output) {  # provide output stream
    out = output;
  }

  proc in1(value1) {  # produce next value for stream 1
    P(empty1); v1 = value1; V(full1);
  }

  proc in2(value2) {  # produce next value for stream 2
    P(empty2); v2 = value2; V(full2);
  }

  process M {
    P(full1); P(full2);  # wait for two input values
    while (v1 != EOS and v2 != EOS)
      if (v1 <= v2)
          { call out(v1); V(empty1); P(full1); }
      else  # v2 < v1
          { call out(v2); V(empty2); P(full2); }
    # consume the rest of the non-empty input stream
    if (v1 == EOS)
      while (v2 != EOS)
          { call out(v2); V(empty2); P(full2); }
    else  # v2 == EOS
      while (v1 != EOS)
          { call out(v1); V(empty1); P(full1); }
    call out(EOS);  # append sentinel
  }
end Merge
```

Figure 8.3 Merge-sort filters using RPC.

```
module Exchange[i = 1 to 2]
  op deposit(int);
body
  int othervalue;
  sem ready = 0;   # used for signaling

  proc deposit(other) {    # called by other module
    othervalue = other;    # save other's value
    V(ready);              # let Worker pick it up
  }
  process Worker {
    int myvalue;
    call Exchange[3-i].deposit(myvalue);  # send to other
    P(ready);             # wait to receive other's value
    ...
  }
end Exchange
```

Figure 8.4 Exchanging values using RPC.

```
module BoundedBuffer
  op deposit(typeT), fetch(result typeT);
body
  process Buffer {
    typeT buf[n];
    int front = 0, rear = 0, count = 0;
    while (true)
      in deposit(item) and count < n ->
            buf[rear] = item;
            rear = (rear+1) mod n; count = count+1;
      [] fetch(item) and count > 0 ->
            item = buf[front];
            front = (front+1) mod n; count = count-1;
      ni
  }
end BoundedBuffer
```

Figure 8.5 Rendezvous implementation of a bounded buffer.

```
module Table
  op getforks(int), relforks(int);
body
  process Waiter {
    bool eating[5] = ([5] false);
    while (true)
      in getforks(i) and not (eating[left(i)] and
           not eating[right(i)] -> eating[i] = true;
      [] relforks(i) ->
            eating[i] = false;
      ni
  }
end Table

process Philosopher[i = 0 to 4] {
  while (true) {
    call getforks(i);
    eat;
    call relforks(i);
    think;
  }
}
```

Figure 8.6 Centralized dining philosophers using rendezvous.

```
module TimeServer
  op get_time() returns int;
  op delay(int);
  op tick();      # called by clock interrupt handler
body TimeServer
  process Timer {
    int tod = 0;  # time of day
    while (true)
      in get_time() returns time -> time = tod;
      [] delay(waketime) and waketime <= tod -> skip;
      [] tick() -> { tod = tod+1; restart timer; }
      ni
  }
end TimeServer
```

Figure 8.7 A time server using rendezvous.

```
module SJN_Allocator
  op request(int time), release();
body
  process SJN {
    bool free = true;
    while (true)
      in request(time) and free by time -> free = false;
      [] release() -> free = true;
      ni
  }
end SJN_Allocator
```

Figure 8.8 Shortest-job-next allocator using rendezvous.

```
optype stream = (int);  # type of data streams

module Merge[i = 1 to n]
  op in1 stream, in2 stream;  # input streams
  op initialize(cap stream);  # link to output stream
body
  process Filter {
    int v1, v2;      # values from input streams
    cap stream out;  # capability for output stream
    in initialize(c) -> out = c ni
    # get first values from input streams
    in in1(v) -> v1 = v; ni
    in in2(v) -> v2 = v; ni
    while (v1 != EOS and v2 != EOS)
      if (v1 <= v2)
          { call out(v1); in in1(v) -> v1 = v; ni }
      else   # v2 < v1
          { call out(v2); in in2(v) -> v2 = v; ni }
    # consume the rest of the non-empty input stream
    if (v1 == EOS)
      while (v2 != EOS)
          { call out(v2); in in2(v) -> v2 = v; ni }
    else  # v2 == EOS
      while (v1 != EOS)
          { call out(v1); in in1(v) -> v1 = v; ni }
    call out(EOS);
  }
end Merge
```

Figure 8.9 Merge sort filters using rendezvous.

```
module Exchange[i = 1 to 2]
  op deposit(int);
body
  process Worker {
    int myvalue, othervalue;
    if (i == 1) {   # one process calls
      call Exchange[2].deposit(myvalue);
      in deposit(othervalue) -> skip; ni
    } else {        # the other process receives
      in deposit(othervalue) -> skip; ni
      call Exchange[1].deposit(myvalue);
    }
    ...
  }
end Exchange
```

Figure 8.10 Exchanging values using rendezvous.

```
module Queue
  op deposit(typeT), fetch(result typeT);
body
  typeT buf[n];
  int front = 1, rear = 1, count = 0;

  proc deposit(item) {
    if (count < n) {
      buf[rear] = item;
      rear = (rear+1) mod n; count = count+1;
    } else
      take actions appropriate for overflow;
  }

  proc fetch(item) {
    if (count > 0) {
      item = buf[front];
      front = (front+1) mod n; count = count-1;
    } else
      take actions appropriate for underflow;
  }
end Queue
```

Figure 8.11 A sequential queue.

```
module BoundedBuffer
  op deposit(typeT), fetch(result typeT);
body
  typeT buf[n];
  int front = 1, rear = 1;
  # local operations used to simulate semaphores
  op empty(), full(), mutexD(), mutexF();
  send mutexD(); send mutexF();
  for [i = 1 to n]  # initialize empty "semaphore"
    send empty();

  proc deposit(item) {
    receive empty(); receive mutexD();
    buf[rear] = item; rear = (rear+1) mod n;
    send mutexD(); send full();
  }

  proc fetch(item) {
    receive full(); receive mutexF();
    item = buf[front]; front = (front+1) mod n;
    send mutexF(); send empty();
  }
end BoundedBuffer
```

Figure 8.12 A bounded buffer using semaphore operations.

```
module ReadersWriters
  op read(result result types);  # uses RPC
  op write(value types);     # uses rendezvous
body
  op startread(), endread();   # local operations
  storage for the database or file transfer buffers;

  proc read(results) {
    call startread();   # get read permission
    read the database;
    send endread();     # release read permission
  }

  process Writer {
    int nr = 0;
    while (true) {
      in startread() -> nr = nr+1;
      [] endread() -> nr = nr-1;
      [] write(values) and nr == 0 ->
            write the database;
      ni
    }
  }
end ReadersWriters
```

Figure 8.13 Readers and writers with encapsulated database.

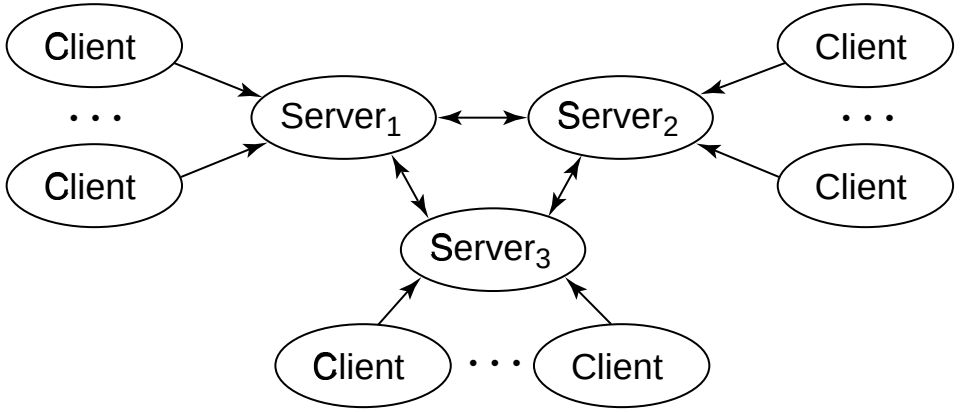


Figure 8.14 Replicated file server interaction pattern.

```
module FileServer[myid = 1 to n]
  type mode = (READ, WRITE);
  op open(mode), close(),       # client operations
     read(result result types), write(value types);
  op startwrite(), endwrite(),  # server operations
     remote_write(value types);
body
  op startread(), endread();   # local operations
  mode use; declarations for file buffers;

  proc open(m) {
    if (m == READ) {
      call startread();    # get local read lock
      use = READ;
    } else {    # mode assumed to be WRITE
      # get write locks for all copies
      for [i = 1 to n]
        call FileServer[i].startwrite();
      use = WRITE;
    }
  }

  proc close() {
    if (use == READ)   # release local read lock
      send endread();
    else   # use == WRITE, so release all write locks
      for [i = 1 to n]
        send FileServer.endwrite()
  }

  proc read(results) {
    read from local copy of file and return results;
  }

  proc write(values) {
    if (use == READ)
      return with error:  file was not opened for writing;
    write values into local copy of file;
    # concurrently update all remote copies
    co [i = 1 to n st i != myid]
      call FileServer[i].remote_write(values);
  }

  proc remote_write(values) { # called by other servers
    write values into local copy of file;
  }
```

```
    process Lock {
      int nr = 0, nw = 0;
      while (true) {
        ## RW: (nr == 0 ∨ nw == 0) ∧ nw <= 1
        in startread() and nw == 0 -> nr = nr+1;
        [] endread() -> nr = nr-1;
        [] startwrite() and nr == 0 and nw == 0 ->
              nw = nw+1;
        [] endwrite() -> nw = nw-1;
        ni
      }
    }
  end FileServer
```

Figure 8.15 Replicated files using one lock per copy.

```
import java.rmi.*;
import java.rmi.server.*;

public interface RemoteDatabase extends Remote {
 public int read() throws RemoteException;
 public void write(int value) throws RemoteException;
}

class Client {
 public static void main(String[] args) {
  try {
   // set the standard RMI security manager
   System.setSecurityManager(new RMISecurityManager());

   // get remote database object
   String name =
       "rmi://paloverde:9999/database";
   RemoteDatabase db =
       (RemoteDatabase) Naming.lookup(name);

   // read command-line argument and access database
   int value, rounds = Integer.parseInt(args[0]);
   for (int i = 0; i < rounds; i++) {
     value = db.read();
     System.out.println("read: " + value);
     db.write(value+1);
   }
  }
  catch (Exception e) {
    System.err.println(e);
  }
 }
}

class RemoteDatabaseServer extends UnicastRemoteObject
          implements RemoteDatabase {
 protected int data = 0;  // the "database"

 public int read() throws RemoteException {
   return data;
 }

 public void write(int value) throws RemoteException {
   data = value;
   System.out.println("new value is: " + data);
 }

 // constructor required because of throws clause
 public RemoteDatabaseServer() throws RemoteException {
   super();
 }
```

```
 public static void main(String[] args) {
  try {
   // create a remote database server object
   RemoteDatabaseServer server =
       new RemoteDatabaseServer();
   // register name and start serving!
   String name =
       "rmi://paloverde:9999/database";
   Naming.bind(name, server);
   System.out.println(name + " is running");
  }
  catch (Exception e) {
    System.err.println(e);
  }
 }
}
```

Figure 8.16 Remote database interface, client, and server.

```
protected type Barrier is
  procedure Arrive;
private
  entry Go;              -- used to delay early arrivals
  count : Integer := 0; -- number who have arrived
  time_to_leave : Boolean := False;
end Barrier;

protected body Barrier is
  procedure Arrive is begin
    count := count+1;
    if count < N then
      requeue Go;   -- wait for others to arrive
    else
      count := count-1; time_to_leave := True;
    end if;
  end;

  entry Go when time_to_leave is begin
    count := count-1;
    if count = 0 then time_to_leave := False; end if;
  end;
end Barrier;
```

Figure 8.17 Barrier synchronization in Ada.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Dining_Philosophers is
  subtype ID is Integer range 1..5;

  task Waiter is             -- Waiter spec
    entry Pickup(I : in ID);
    entry Putdown(I : in ID);
  end
  task body Waiter is separate;

  task type Philosopher is   -- Philosopher spec
    entry init(who : ID);
  end;

  DP : array(ID) of Philosopher;  -- the philosophers
  rounds : Integer;               -- number of rounds

  task body Philosopher is   -- Philosopher body
    myid : ID;
  begin
    accept init(who); myid := who; end;
    for j in 1..rounds loop
      -- "think"
      Waiter.Pickup(myid);  -- pick forks up
      -- "eat"
      Waiter.Putdown(myid); -- put forks down
    end loop;
  end Philosopher;

begin  -- read in rounds, then start the philosophers
  Get(rounds);
  for j in ID loop
    DP(j).init(j);
  end loop;
end Dining_Philosophers;
```

Figure 8.18 Dining philosophers in Ada: Main program.

```
separate (Dining_Philosophers)
task body Waiter is
  entry Wait(ID);     -- used to requeue philosophers
  eating : array (ID) of Boolean; -- who is eating
  want : array (ID) of Boolean;   -- who wants to eat
  go : array(ID) of Boolean;      -- who can go now
begin
  for j in ID loop           -- initialize the arrays
    eating(j) := False; want(j) := False;
  end loop;
  loop             -- basic server loop
   select
     accept Pickup(i : in ID) do  -- DP(i) needs forks
       if not(eating(left(i)) or eating(right(i))) then
         eating(i) := True;
       else
         want(i) := True; requeue Wait(i);
       end if;
     end;
   or
     accept Putdown(i : in ID) do  -- DP(i) is done
       eating(i) := False;
     end;
     -- check neighbors to see if they can eat now
     if want(left(i)) and not eating(left(left(i))) then
       accept Wait(left(i));
       eating(left(i)) := True; want(left(i)) := False;
     end if;
     if want(right(i)) and not eating(right(right(i)))
       then  accept Wait(right(i));
       eating(right(i)) := True; want(right(i)) := False;
     end if;
   or
     terminate;  -- quit when philosophers have quit
   end select;
  end loop;
end Waiter;
```

Figure 8.19 Dining philosophers in Ada: Waiter task.

```
global CS
  op CSenter(id: int) {call}    # must be called
  op CSexit()    # may be invoked by call or send
body CS
  process arbitrator
    do true ->
      in CSenter(id) by id ->
          write("user", id, "in its CS at", age())
      ni
      receive CSexit()
    od
  end
end

resource main()
  import CS
  var numusers, rounds: int
  getarg(1, numusers); getarg(2, rounds)

  process user(i := 1 to numusers)
    fa j := 1 to rounds ->
      call CSenter(i)         # enter critical section
      nap(int(random(100)))   # delay up to 100 msec
      send CSexit()           # exit critical section
      nap(int(random(1000))) # delay up to 1 second
    af
  end
end
```

Figure 8.20 An SR program to simulate critical sections.

```
accept E(formals) do
 statement list;
end;
```

```
select when B1 => accept statement; additional statements;
or ...
or     when Bn => accept statement; additional statements;
end select;
```

```
select  entry call; additional statements;
else    statements;
end select;
```

```
select  entry call; additional statements;
or      delay statement; additional statements;
end select;
```

Ada accept and select statements.

```
protected type Name is
  function, procedure, or entry declarations;
private
  variable declarations;
end Name;


protected body Name is
  function, procedure, or entry bodies;
end Name;


requeue Opname;
```

Ada protected types and requeue statement.

```
task Name is
 entry declarations;
end;
```

```
entry Identifier(formals);
```

```
task body Name is
 local declarations;
begin
 statements;
end Name;
```

```
call T.E(actuals);
```

Ada tasks, entries, and call statements.

```
module mname
   headers of exported operations;
body
   variable declarations;
   initialization code;
   procedures for exported operations;
   local procedures and processes;
end mname


op opname(formals) [returns result]


proc opname(formal identifiers) returns result identifier
   declarations of local variables;
   statements
end


call mname.opname(arguments)
```

Modules: declarations, operations, procs, and calls.

```
op opname(types of formals);
```

```
in opname(formal identifiers) -> S; ni
```

```
in op1(formals1) and B1 by e1 -> S1;
[] ...
[] opn(formalsn) and Bn by en -> Sn;
ni
```

Rendezvous: operations and input statements.

```
resource name    # specification part
   import clauses
   operation and type declarations
body name(formals)  # body
   variable and other local declarations
   initialization code
   procedures and processes
   finalization code
end name
```

```
rcap := create name(actuals)
```

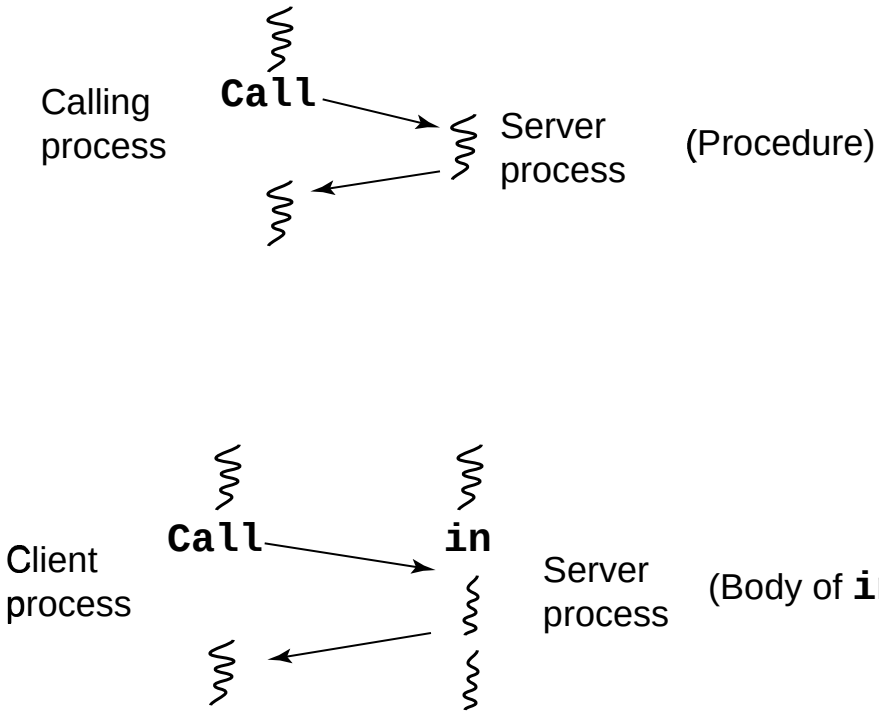
```
vmcap := create vm() on machine
```

```
resource silly()
  write("Hello world.")
  final
    write("Goodbye world.")
  end
end
```

SR: resources and create statements.

invocation	*service*	*effect*
`call`	**`proc`**	procedure call
`call`	**`in`**	rendezvous
`send`	**`proc`**	dynamic process creation
`send`	**`in`**	asynchronous message passing

Multiple primitives: ways to invoke and service operations.



Timing and flow of control for RPC (top) and rendezvous (bottom).