RPC and Rendezvous

INF4140

01.11.12

Lecture 9

Outline

More on asynchronous message passing

- Interacting processes with different patterns of communication
- Summary

Remote procedure call (RPC)

- What is RPC
- Example: time server

Rendezvous

- What is rendezvous
- Examples: buffer, time server

Combinations of RPC, rendezvous and message passing

- Examples: bounded buffer, readers/writers

Interacting peers (processes): exchanging values example

Look at processes as peers.

Example: Exchanging values

- Consider n processes P[0], ..., P[$n-1$], $n > 1$
- Every process has a number – stored in a local variable v
- Goal: all processes knows the largest and smallest number.

Look at different patterns of communication:

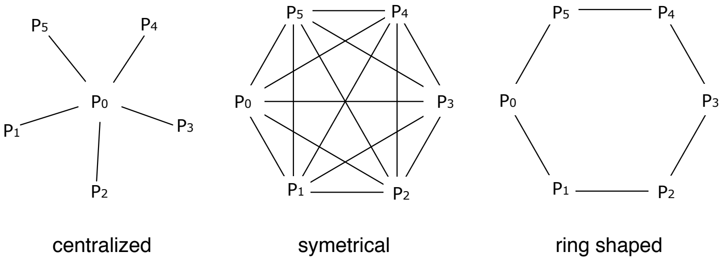
Interacting peers (processes): exchanging values example

Look at processes as peers.

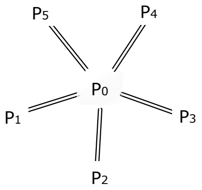
Example: Exchanging values

- Consider n processes P[0], ..., P[$n-1$], $n > 1$
- Every process has a number – stored in a local variable v
- Goal: all processes knows the largest and smallest number.

Look at different patterns of communication:



Centralized solution



Process P[0] is the coordinator process:

- P[0] does the calculation
- The other processes sends their values to P[0] and waits for a reply.

Number of messages: (Just count the number of send:)

$$\text{P[0]:} \quad n-1$$
$$\text{P[1]}, \ldots, \text{P}[n-1]\text{:} \quad (n-1)\times 1$$

Total: $(n-1)+(n-1) = 2(n-1)$ messages

Number of channels: n

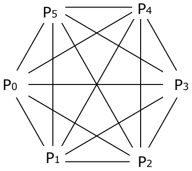
Centralized solution: code

```
chan values(int),
     results[1..n-1](int smallest, int largest);

process P[0] {  # coordinator process
  int v = ...;
  int new, smallest = v, largest = v; # initialization
  # get values and store the largest and smallest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)       smallest = new;
    if (new > largest)        largest = new;
  }
  # send results
  for [i = 1 to n-1]
    send results[i](smallest, largest);
}
process P[i = 1 to n-1] {
  int v = ...;
  int smallest, largest;

  send values(v);
  receive results[i](smallest, largest);}
# Fig. 7.11 in Andrews (corrected a bug)
```

Symmetrical solution



"Single-programme, multiple data (SPMD)"-solution:

Each process executes the same code
and shares the results with all other processes.

Number of messages:
n processes sending $n-1$ messages each,
Total: $n(n-1)$ messages.

Number of channels: n

Symmetrical solution: code

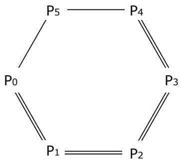
```
chan values[n](int);

process P[i = 0 to n-1] {
  int v = ...;
  int new, smallest = v, largest = v;

  # send v to all n-1 other processes
  for [j = 0 to n-1 st j != i]
    send values[j](v);

  # get n-1 values
  # and store the smallest and largest.
  for [j = 1 to n-1] { # j not used in the loop
    receive values[i](new);
    if (new < smallest)     smallest = new;
    if (new > largest)      largest = new;
  }
} # Fig. 7.12 from Andrews
```

Ring solution



Almost symmetrical, except P[0], P[$n-2$] and P[$n-1$].

Each process executes the same code
and sends the results to the next process (if necessary).

Number of messages:

P[0]:	2
P[1], ..., P[$n-3$]:	$(n-3) \times 2$
P[$n-2$]:	1
P[$n-1$]:	1

$2 + 2(n-3) + 1 + 1 = 2(n-1)$ messages sent.

Number of channels: n .

Ring solution: code (1)

```
chan values[n](int smallest, int largest);

process P[0] { # starts the exchange
  int v = ...;
  int smallest = v, largest = v;
  # send v to the next process, P[1]
  send values[1](smallest, largest);
  # get the global smallest and largest from P[n-1]
  #    and send them to P[1]
  receive values[0](smallest, largest);
  send values[1](smallest, largest);
}
```

Ring solution: code (2)

```
process P[i = 1 to n-1] {
  int v = ...;
  int smallest, largest;
  # get smallest and largest so far,
  #     and update them by comparing them to v
  receive values[i](smallest, largest)
  if (v < smallest) smallest = v;
  if (v > largest)  largest = v;
  # forward the result, and wait for the global result
  send values[(i+1) mod n](smallest, largest);
  if (i < n-1)
    receive values[i](smallest, largest);
  # forward the global result, but not from P[n-1] to P[0]
  if (i < n-2)
    send values[i+1](smallest, largest);
} # Fig. 7.13 from Andrews (modified)
```

Message passing: Sumary

Message passing is well suited to programming filters and interacting peers (where processes communicates one way by one or more channels).

May be used for client/server applications, but:

- Each client must have its own reply channel
- In general: two way communication needs two channels

$\Rightarrow$ many channels

RPC and rendezvous are better suited for client/server applications.

Remote Procedure Call: main idea

CALLER

at computer A

CALLEE

at computer B

```
                                    op foo(FORMALS); # declaration

...
call foo(ARGS);        ----->       proc foo(FORMALS)  # new process
                                      ...
                       <-----       end;
...
```

RPC (cont.)

RPC combines some elements from monitors and message passing

- As ordinary procedure call, but
  caller and callee may be on different machines.
- Caller is blocked until the called procedure is done, as with monitor calls and synchronous message passing.
  Asynchronous programming is not supported directly.[1]
- A new process handles each call.
- Potentially two way communication: caller sends arguments and receives return values.

[1] But in Creol . . .

RPC: module, procedure, process

Module: new program component – contains both

- procedures and processes.

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

Modules may be executed on different machines

M has: Procedures and processes

- may share variables
- execute concurrently ⇒ *must be synchronized to achieve mutex*
- May only communicate with processes in M' by procedures exported by M'

RPC: operations

Declaration of operation O:

op O(*formal parameters*.) [returns *result*] ;

Implementation of operation O:

proc O(*formal identifiers*.) [returns *result identifier*]{
  declaration of local variables;
  statements
}

Call of operation O in module M:

call M.O(*arguments*)

Processes: as before.

Example: Time server (RPC)

Problem: Implement a module that provides timing services to processes in other modules.

The time server defines two visible operations:

- get_time() returns int – returns time of day
- delay(int interval) – let the caller sleep a given number of time units

Multiple clients may call get_time and delay at the same time
⇒ Need to protect the variables.

The time server has an internal process that gets interrupts from a machine clock and updates tod.

Time server: code (RPC 1)

```
module TimeServer
  op get_time() returns int;
  op delay(int interval);
body
  int tod = 0;              # time of day
  sem m = 1;                # for mutex
  sem d[n] = ([n] 0);  # for delayed processes
  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(int interval) {
    P(m);           # assume unique myid and i [0,n-1]
    int waketime = tod + interval;
      insert  (waketime, myid) at appropriate place in napQ;
    V(m);
    P(d[myid]);  # Wait to be awoken
  }
  process Clock ...
  .
  .
  .
end TimeServer
```

Time server: code (RPC 2)

```
process Clock {
    int id; start hardware timer;
    while (true) {
      wait for interrupt, then restart hardware timer
      tod = tod + 1;
      P(m);                                              # mutex
      while (tod >=  smallest waketime on napQ) {
        remove  (waketime, id) from napQ;
        V(d[id]);                                        # awake process
      }
      V(m);                                              # mutex
  } }
end TimeServer # Fig. 8.1 of Andrews
```

Rendezvous

RPC:

- Offers inter module communication
- Synchronization must be programmed explicitly

Rendezvous:

- Known from the language Ada (US DoD)
- Combines communication and synchronization between processes
- *No* new process made when a call is made.
  Does 'rendezvous' with existing process
- Operations are executed one at the time

synch_send and receive may be considered as primitive rendezvous.

Rendezvous: main idea

CALLER

at computer A

CALLEE

at computer B

```
                                                op foo(FORMALS); # declaration

...                                             ... # existing process
call foo(ARGS);           ----->                in foo(FORMALS)  ->
                                                  BODY;
                          <-----                ni
...
```

Rendezvous: module declaration

```
module M
  op O1(types);
  ...
  op On(types);
body

  process P1 {
    variable declarations;
    while (true)
      in O1(formals) and B1 -> S1;
      ...
      [] On (formals) and Bn -> Sn;
      ni
  }
  ... other processes
end M
```

Rendezvous: syntax of expressions

Call:

```
call Oi (expr1, ..., exprm);
```

$\texttt{call}\ O_i\ (expr_1, \ldots, expr_m);$

Input statement, multiple guarded expressions:

$\texttt{in}\ O_1(v_1, \ldots v_{m_1})\ \texttt{and}\ B_1 \texttt{->} S_1;$
$\ldots$
$[\,]\ O_n(v_1, \ldots v_{m_n})\ \texttt{and}\ B_n \texttt{->} S_n;$
$\texttt{ni}$

The guard consists of:

- and B_i – synchronization expression (optional)
- S_i – statements (one or more)

The variables $v_1, \ldots, v_{m_i}$ may be referred by B_i and S_i may read/write to them.

Rendezvous: semantics of input statement

Consider the following:

```
in ...
[] O_i(v_i, ..., v_{m_i}) and B_i -> S_i ;
...
ni
```

The guard *succeeds* when O_i is called and B_i is true (or omitted).

Execution of the in statement:

- Delays until a guard succeeds
- If more than one guard succeed, the oldest call is served
- Values are returned to the caller
- The the call- and in statements terminates

Example: bounded buffer (rendezvous)

```
module BoundedBuffer
  op deposit(elem), fetch(result elem);
body
  process Buffer {
    elem buf[n];
    int front = 0, rear = 0, count = 0;
    while (true)
      in deposit(item) and count < n ->
            buf[rear] = item; count++;
            rear = (rear+1) mod n;
      [] fetch(item) and count > 0 ->
            item = buf[front]; count--;
            front = (front+1) mod n;
      ni
  }
end BoundedBuffer  # Fig. 8.5 of Andrews
```

Example: time server (rendezvous)

```
module TimeServer
  op get_time() returns int;
  op delay(int);    # Waketime as argument
  op tick();        # called by the clock interrupt handler
body
  process Timer {
    int tod = 0;
    start timer;
    while (true)
      in get_time() returns time -> time = tod;
      [] delay(waketime) and waketime <= tod -> skip;
      [] tick() -> { tod++; restart timer;
      ni
  }
end TimeServer  # Fig. 8.7 of Andrews
```

RPC, rendezvous and message passing

We do now have several combinations:

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

RPC, rendezvous and message passing

We do now have several combinations:

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

in addition (not in Andrews)

- asynchronous procedure call, wait-by-necessity, futures

Rendezvous, message passing and semaphores

Comparing input statements and receive:

in O(a_1, ...,a_n) -> v_1=a_1,...,v_n=a_n ni $\iff$ receive O(v_1, ..., v_n)

Comparing message passing and semaphores:

send O() and receive O() $\iff$ V(O) and P(O)

Example: Bounded buffer (again)

```
module BoundedBuffer
  op deposit(elem), fetch(result elem);
body
  elem buf[n];
  int front = 0, rear = 0;
  # local operation to simulate semaphores
  op empty(), full(), mutexD(), mutexF();
  send mutexD(); send mutexF();   # init. "semaphores" to 1
  for [i = 1 to n]                # init. empty-"semaphore" to n
    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  # Fig. 8.12 of Andrews
```

The primitive $?O$ in rendezvous

New primitive on operations, similar to empty(. . .) for condition variables and channels.

$?O$ means number of pending invocations of operation O.

Useful in the input statement to give priority:

in O_1 . . . -> S_1;
[ ] O_2 . . . and $?O_1 == 0$ -> S_2;
ni

Here O_1 has a higher priority than O_2.

Readers and writers

```
module ReadersWriters
  op read(result types);   # uses RPC
  op write(types);         # uses rendezvous
body
  op startread(), endread();  # local ops.
  ... database (DB)...;

  proc read(vars) {
    call startread();      # get read access
    ... read vars from DB ...;
    send endread();        # free DB
  }
  process Writer {
    int nr = 0;
    while (true)
      in startread() -> nr++;
      [] endread()   -> nr--;
      [] write(vars) and nr == 0 ->
           ... write vars to DB ... ;
      ni
  }
end ReadersWriters
```

Readers and writers: prioritize writers

```
module ReadersWriters
  op read(result types);   # uses RPC
  op write(types);         # uses rendezvous
body
  op startread(), endread();  # local ops.
  ... database (DB)...;

  proc read(vars) {
    call startread();      # get read access
    ... read vars from DB ...;
    send endread();        # free DB
  }
  process Writer {
    int nr = 0;
    while (true)
      in startread() and ?write == 0 -> nr++;
      [] endread()    -> nr--;
      [] write(vars) and nr == 0 ->
           ... write vars to DB ... ;
      ni
  }
end ReadersWriters
```