# INF4140 - Models of concurrency
## RPC and Rendezvous

INF4140

28 Oct. 2013

RPC and Rendezvous

## Outline

- More on asynchronous message passing
  - interacting processes with different patterns of communication
  - Summary
- Remote procedure calls
  - What is RPC
  - Examples: time server, merge filters, exchanging values
- Rendez-vous
  - What is rendez-vous?
  - Examples: buffer, time server, exchanging values
- Combinations of RPC, rendezvous and message passing
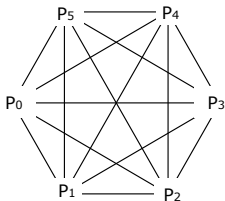  - Examples: bounded buffer, readers/writers

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.
- simplistic problem, but "characteristic" of distributed computation and info-distribution
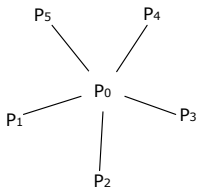
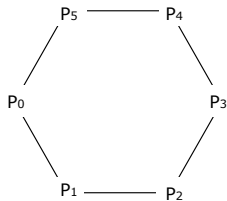centralized     symetrical     ring shaped

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.
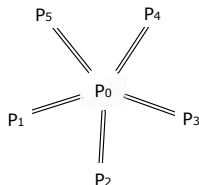
Number of *messages:* (number of send:)

$$\begin{array}{ll} \text{P[0]:} & n-1 \\ \text{P[1]}, \ldots, \text{P[}n-1\text{]:} & (n-1) \\ \text{Total: } (n-1)+(n-1) = 2(n-1) \text{ messages} \end{array}$$

Number of *channels:* $n$[1]

---

[1]not good style here

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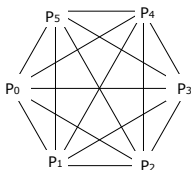
"Single-programme, multiple data (SPMD)"-solution:

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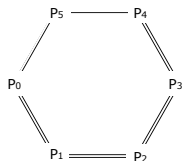
Almost symmetrical, except P[0], P[$n-2$] and P[$n-1$].

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

Number of messages:

$$\begin{array}{rc} \text{P[0]:} & 2 \\ \text{P[1]}, \ldots, \text{P[}n-3\text{]:} & (n-3) \times 2 \\ \text{P[}n-2\text{]:} & 1 \\ \text{P[}n-1\text{]:} & 1 \end{array}$$

$2 + 2(n-3) + 1 + 1 = 2(n-1)$ 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: 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

⇒ 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: combines elements from monitors and message passing

- As ordinary procedure call, but caller and callee may be on different machines.[2]

- Caller: blocked until called procedure is done, as with monitor calls and synchronous message passing.

- Asynchronous programming: not supported directly.

- A new process handles each call.

- Potentially two way communication: caller sends arguments and receives return values.

---

[2]RMI

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 $\Rightarrow$ *must be synchronized to achieve mutex*
- May only communicate with processes in $M'$ by procedures exported by $M'$

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.

- RPC: primarily a *communication* mechanism
- within the module: in principle allowed:
    - more than one process
    - shared data
- ⇒ need for synchronization
- two approaches
    1. "implicit":
        - as in monitors: mutex built-in
        - additionally condition variables (or semaphores)
    2. "explicit":[3]
        - user-programmed mutex and synchronization (like semaphorse, local monitors etc)

---

[3]assumed in the following

- module providing timing services to processes in other modules.
- interface: 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.
- internal process that gets interrupts from machine clock and updates tod.

```
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
```

```
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
```

RPC:

- offers inter-module communication
- synchronization (often): must be programmed explicitly

Rendezvous:

- Known from the language Ada (US DoD)
- Combines communication and synchronization between processes
- *No* new process created for each call
- instead: perform 'rendezvous' with existing process
- Operations are executed one at the time

synch_send and receive may be considered as primitive rendezvous.

**CALLER**

at computer **A**

**CALLEE**

at computer **B**

```
                                op foo(FORMALS); # declaration

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

```
module M
  op O_1(types);
  ...
  op O_n(types);
body

  process P_1 {
    variable declarations;
    while (true)
      in O_1(formals) and B_1 -> S_1;
      ...
      [] O_n (formals) and B_n -> S_n;
      ni
  }
  ... other processes
end M
```

Call:

```
call O_i (expr_1,...,expr_m);
```

Input statement, multiple guarded expressions:

```
in  O_1(v_1,...v_{m_1}) and  B_1 -> S_1;
    ...
[]  O_n(v_1,...v_{m_n}) and  B_n -> S_n;
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.

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

- different versions of rendezvous, depending on the language
- origin: ADA (accept-statement) (see [And00, Section 8.6])
- design variation points
  - synchronization expressions or not?
  - scheduling expressions or not
  - can the guard expect the *values* for input variables or not?
  - non-determinism
  - checking for *absence* of messages? priority
  - checking in more than one operation?

```
module BoundedBuffer
  op deposit(TypeT), fetch(result TypeT);
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
```

```
module TimeServer
  op get_time() returns int;
  op delay(int);    # absolute 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
```

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 |

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

Comparing input statements and receive:

in $O(a_1, \ldots, a_n)$ -> $v_1 = a_1, \ldots, v_n = a_n$ ni $\quad \Longleftrightarrow \quad$ receive $O(v_1, \ldots, v_n)$

Comparing message passing and semaphores:

send $O()$ and receive $O()$ $\quad \Longleftrightarrow \quad$ $V(O)$ and $P(O)$

```
module BoundedBuffer
  op deposit(typeT), fetch(result typeT);
body
  elem buf[n];
  int front = 0, rear = 0;
  # local operation to simulate semaphores
  op empty(), full(), mutexD(), mutexF();            // operations
  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
```

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
     O₁  ...                   -> S₁;
[ ]
     O₂  ...  and  (?O₁ = 0)  -> S₂;

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 typeT);   # uses RPC
  op write(typeT);         # 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
```

[And00]   Gregory R. Andrews.

          *Foundations of Multithreaded, Parallel, and Distributed Programming*.

          Addison-Wesley, 2000.