Concurrent Programming (RIO) 14.3.2011

Lecture 9: Channels and RPC

Concurrency Control in Distributed Environment

Ch 8 [BenA 06]

Messages
Channels
Rendezvous
RPC and RMI

Distributed System

- No shared memory
- Communication with messages
- Tightly coupled systems
  - Processes alive at the same time
- Persistent systems
  - Data stays even if processes die
- Fully distributed systems
  - Everything goes
Communication with Messages (4)

**Sender, receiver**

**Synchronous/asynchronous communication**

Message Passing

- **Synchronous communication**
  - Atomic action
  - Both wait until communication complete
- **Asynchronous communication**
  - Sender continues after giving the message to OS for delivery
  - May get an acknowledgement later on
    - Message received or not
- **Addressing**
  - Some address for receiver process
    - Process name, id, node/name, …
  - Some address for the communication channel
    - Port number, channel name, …
  - Some address for requested service
    - Broker will find out, sooner or later
      - After message has been sent?
    - Service address not known at service request time

14.3.2011 Copyright Teemu Kerola 2011
Synchronization levels

Process A
- \( X=f(\cdot \cdot) \)
- send \( X \) to \( B \)
- \( X: 10 \)
- OS kernel

Process B
- receive \( X \) from \( A \)
- \( Y=f(X) \)
- \( X: 5 \)
- OS kernel

Synchronous?

Reliable comm

Asynchronous?
Synchronization levels (2/5)

Process A

asynchronous?

... X=f(..); send X to B ...

X: 10

OS kernel send Data Com

Process B

... receive X from A

Y=f(X);

... X: 5

DC receive OS kernel

Synchronization levels (3/5)

Process A

asynchronous?

... X=f(..); send X to B ...

X: 10

OS kernel send DC

Process B

... receive X from A

Y=f(X);

... X: 5

DC receive OS kernel
Synchronization levels (4/5)

**Process A**

```
X := f(..); send X to B
X := 10
OS kernel send DC
```

**Process B**

```
... receive X from A
Y := f(X);
... send X to B
X := 5
OS kernel receive DC
```

Synchronization levels (5/5)

**Process A**

```
X := f(..); send X to B
X := 10
OS kernel send DC
```

**Process B**

```
... receive X from A
Y := f(X);
... send X to B
X := 5
OS kernel receive DC
```

synchronous?
Message Passing

- Symmetric communication
  - Cooperating processes at same level
  - Both know about each others address
  - Communication method for a fixed channel

- Asymmetric communication
  - Different status for communicating processes
  - Client-server model
    - Server address known, client address given in request

- Broadcast communication
  - Receiver not addressed directly
  - Message sent to everybody (in one node?)
  - Receivers may be limited in number
    - Just one?
    - Only the intended recipient(s) will act on it?

Wait Semantics

- Sender
  - Continue after OS has taken the message
    - Non-blocking send
  - Continue after message reached receiver node
    - Blocking send
  - Continue after message reached receiver process
    - Blocking send

- Receiver
  - Continue only after message received
    - Blocking receive
  - Continue even if no message received
    - Status indicated whether message received or not
    - Non-blocking receive
Message Passing

- Data flow
  - One-way
    - Synchronous may be one-way
    - Asynchronous is always one-way
  - Two-way
    - Synchronous may be two-way
    - Two asynchronous communications

- Primitives
  - One message at a time
  - Need addresses for communicating processes
  - Operating system level service
  - Usually not programming language level construct
    - Too primitive: need to know node id, process id, port number, …
Channels

- History of languages utilizing channels
  - Guarded Commands
    - Dijkstra, 1975
  - Communicating Sequential Processes
    - CSP, Hoare, 1978
  - Occam
    - David May et al, 1983
    - Hoare as consultant
    - Inmos Transputer

Guarded Commands (Dijkstra)

- Way to describe predicate transformer semantics
- Communication not really specified
- Guarded command
  - Condition or guard
  - Statement

\[
\begin{align*}
x, y = X, Y & \quad \text{-- statement (unguarded)} \\
\text{do} & \quad \text{-- loop command, loop terminates when } x = y \\
& x \neq y \quad \text{if} \quad \text{-- conditional command (itself guarded)} \\
& x > y \quad x := x - y \quad \text{-- guarded statement in the if} \\
& y > x \quad y := y - x \\
\text{fi} \\
\text{od} \\
\text{print } x ; & \quad \text{-- another statement, also unguarded}
\end{align*}
\]
Communicating Sequential Processes – CSP (Hoare)

- **Language** for modeling and analyzing the behavior of concurrent communicating systems
- A known group of processes A, B, ...
- Communication:
  - output statement: B!e
    - evaluate e, send the value of e to B
  - input statement: A?x
    - receive the value from A to x
  - input, output: blocking statements
  - output & input: “distributed assignment”
    - Communicate value from one process to a variable in some other process

CSP communication

- **Input/output statements**
  - Destination!port (e_1, ..., e_n);
  - Source?port (x_1, ..., x_n);
- **Binding**
  - Communication with **named processes**
  - Matching types for communication
- **Example:** Copy (West => Copy => East)

West:

```plaintext
do true ->
Copy!c;
...
od
```

Copy:

```plaintext
do true ->
West?c;
East!c;
od
```

East:

```plaintext
do true ->
Copy?c;
...
od
```
OCCAM Language

- Communication through **named channels**
  - Globally defined
    - Somewhere, in advance
  - Each channel has one sender and one receiver
    - Both processes in some nodes
- Transputer
  - Multicomputer
    - E.g., 100 node Hathi-2 in ÅA
  - Automatic message routing for channels
  - Programmed with OCCAM

OCCAM Example

(Andrews, p 331)

```
PROC Copy (CHAN OF BYTE West, EAsks, East)
BYTE c1, c2, dummy; -- buffer size = 2
SEQ
West ? c1                      - - West has 1st byte
WHILE TRUE
ALT
West ? c2            - - West has new byte
SEQ
East ! c1    - - send previous byte
c1 := c2    - - copy to buffer c1
EAsks ? dummy  - - East wants a byte
SEQ
East ! c1    - - send previous byte
West ? c1  - - receive next one
```

- How to bind processes to nodes? 8 vs. 100 nodes?
- How to bind channels to processes, physical system?
  - 4 physical ports (N, S, E, W) in each processor
Channels

- Communication through named channels
  - Typed, global to processes
  - Programming language concept
  - Any one can read/write
    (usually limited in practice)
- Pipe or mailbox
- Synchronous, one-way (?)
- How to tie in with many nodes?
  - Not really thought through! Easy with shared memory!

Algorithm 8.1: Producer-consumer (channels)

channel of integer ch

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer x</td>
<td>integer y</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: x ← produce</td>
<td>q1: ch = y</td>
</tr>
<tr>
<td>p2: ch ← x</td>
<td>q2: consume(y)</td>
</tr>
</tbody>
</table>

buffer size?
Filtering Problem

- Compress many (at most MAX) similar characters to pairs ...
  - \{nr of chars, char\}
- \ldots and place newline ('\n') after every K\textsuperscript{th} character in the compressed string
- Why is it called “Conway’s problem”?
  - “Classic coroutine example”

---

**Algorithm 8.2: Conway’s problem**

constant integer MAX ← 9
constant integer K ← 4
channel of integer inC, pipe, outC

<table>
<thead>
<tr>
<th>compress</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{char } c, \text{ previous } \leftarrow 0)</td>
<td>(\text{char } c)</td>
</tr>
<tr>
<td>integer (n \leftarrow 0)</td>
<td>integer (m \leftarrow 0)</td>
</tr>
<tr>
<td>(\text{\texttt{inC }\leftarrow \text{previous}})</td>
<td>loop forever</td>
</tr>
<tr>
<td>(\text{\texttt{no last char?}})</td>
<td>(\text{\texttt{loop forever}})</td>
</tr>
<tr>
<td>p1: (\text{\texttt{inC }\leftarrow c})</td>
<td>q1: (\text{\texttt{pipe }\leftarrow c})</td>
</tr>
<tr>
<td>p2: if ((c = \text{previous}) \text{ and } (n &lt; \text{MAX } - 1))</td>
<td>q2: (\text{\texttt{outC }\leftarrow c})</td>
</tr>
<tr>
<td>p3: (n \leftarrow n + 1)</td>
<td>q3: (m \leftarrow m + 1)</td>
</tr>
<tr>
<td>else</td>
<td>q4: if (m \geq K)</td>
</tr>
<tr>
<td>p4: if (n &gt; 0)</td>
<td>q5: (\text{\texttt{outC }\leftarrow \text{newline}})</td>
</tr>
<tr>
<td>p5: (\text{\texttt{pipe }\leftarrow \text{\texttt{intToChar(n+1)}}})</td>
<td>q6: (m \leftarrow 0)</td>
</tr>
<tr>
<td>p6: (n \leftarrow 0)</td>
<td>q7:</td>
</tr>
<tr>
<td>p7: (\text{\texttt{pipe }\leftarrow \text{previous}})</td>
<td>q8:</td>
</tr>
<tr>
<td>p8: (\text{previous }\leftarrow c)</td>
<td></td>
</tr>
</tbody>
</table>

---

Matrix Multiplication with Channels

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\times
\begin{bmatrix}
1 & 0 & 2 \\
0 & 1 & 2 \\
1 & 0 & 0
\end{bmatrix}
= 
\begin{bmatrix}
4 & 2 & 6 \\
10 & 5 & 18 \\
16 & 8 & 30
\end{bmatrix}
\]

- \(16 = (7 \times 8 \times 9) \times (1 \times 0 \times 1)\)
- \(30 = (7 \times 8 \times 9) \times (2 \times 2 \times 0)\)
- Process for every multiply-add

\[7 \times 2 + 8 \times 2 + 9 \times 0 + 0 = 30\]

Process Array for Matrix Multiplication

- 27 processes
- 24 channels
- Column 1 contains 1 row, sends it down one element at a time
- West-bound multiply-add, South-bound copy North
- Contains 1 value, makes three multiply-adds, forwards values down

How to initialize everything?

How to synchronize everything?
Algorithm 8.3: Multiplier process with channels

```plaintext
Algorithm 8.3: Multiplier process with channels

integer FirstElement
<channel of integer North, East, South, West
integer Sum, integer SecondElement

loop forever
p1: North ⇒ SecondElement ← wait 1st for this (*)
p2: East ⇒ Sum ← and then for this
p3: Sum = Sum + FirstElement • SecondElement
p4: South = SecondElement
p5: West = Sum
```

• How to map processes to nodes?
• How to map channels to processes?
  - North channel of one process the South channel of some other
• North-South data flow has priority (*
  - Waiting even when data-flow East-West available
  - Node on East may be blocked unnecessarily

Algorithm 8.4: Multiplier with channels and selective input

```plaintext
Algorithm 8.4: Multiplier with channels and selective input

integer FirstElement
<channel of integer North, East, South, West
integer Sum, integer SecondElement

loop forever
either
p1: North ⇒ SecondElement
p2: East ⇒ Sum

or
p3: East ⇒ Sum
p4: North ⇒ SecondElement
p5: South = SecondElement
p6: Sum = Sum + FirstElement • SecondElement
p7: West = Sum
```

• Guarded statement
  - Execute one selective input statement
    • Nondeterministic selection (if both available)
    • p2 follows p1, it does not compete with p3

14.3.2011 Copyright Teemu Kerola 2011
Dining Philosophers with Channels

- Each *fork* $i$ is a process, $\text{forks}[i]$ is a channel
- Each *philosopher* $i$ is a process

**Algorithm 8.5: Dining philosophers with channels**

<table>
<thead>
<tr>
<th>philosopher $i$</th>
<th>fork $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean dummy</td>
<td>boolean dummy</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: think</td>
<td>q1: $\text{forks}[i] \Leftarrow true$</td>
</tr>
<tr>
<td>p2: $\text{forks}[i] \Rightarrow \text{dummy}$</td>
<td>q2: $\text{forks}[i] \Rightarrow \text{dummy}$</td>
</tr>
<tr>
<td>p3: $\text{forks}[^0][i] \Rightarrow \text{dummy}$</td>
<td>q3:</td>
</tr>
<tr>
<td>p4: eat</td>
<td>q4:</td>
</tr>
<tr>
<td>p5: $\text{forks}[i] \Leftarrow \text{true}$ (would false)</td>
<td>q5:</td>
</tr>
<tr>
<td>p6: $\text{forks}[^0][i] \Leftarrow \text{true}$ be ok?</td>
<td>q6:</td>
</tr>
</tbody>
</table>

- Would it be enough to initialize each $\text{forks}[i] \Leftarrow true$?
  - Do you really need $\text{forks}[i] \Rightarrow \text{dummy}$ in fork $i$? Why?

14.3.2011 Copyright Teemu Kerola 2011
**Rendezvous** (1978, Abrial & Andrews)

- **Synchronization with communication**
  - No channels, usage similar to procedure calls
  - One (accepting) process waits for one of the (calling) processes
    - One request in service at a time
  - Calling process must know id of the accepting process
  - Accepting process does not need to know the id of calling process
  - May involve parameters and return value

- **Good for client-server synchronization**
  - Clients are calling processes
  - Server is accepting process
  - Server is active process
  - Language construct, no mapping for real system nodes

---

**Algorithm 8.6: Rendezvous**

<table>
<thead>
<tr>
<th>client</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer parm, result</td>
<td>integer p, r</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: parm \leftarrow \ldots</td>
<td>q1:</td>
</tr>
<tr>
<td>p2: server.service(parm, result)</td>
<td>q2: accept service(p, r)</td>
</tr>
<tr>
<td>p3: use(result)</td>
<td>q3: r \leftarrow do the service(p)</td>
</tr>
</tbody>
</table>

- Can have many similar clients
- Implementation with messages (e.g.)
  - Service request in one message
    - Arguments must be marshalled
      (make them suitable for transmission)
  - Wait until reply received
  - Reply result in another message
Guards in Rendezvous

- Additional constraint for accepting given service call
- Accept service call, if
  - Someone requests it and
  - Guard for that request type is true
- Guard is based on local state
- If many such requests (with open guards) available, select one randomly
- Complete one request at a time
  - Implicit mutex

Ada Rendezvous

Bounded Buffer in Ada

```ada
begin
  loop
    select
      when Count < Index'Last =>
        accept Append(I: in Integer) do
          B(In_Ptr) := I;
          end Append;
          Count := Count + 1; In_Ptr := In_Ptr + 1;
      or when Count > 0 =>
        accept Take(I: out Integer) do
          l := B(Out_Ptr);
          end Take;
          Count := Count - 1; Out_Ptr := Out_Ptr + 1;
      or terminate
    end select;
  end loop;
end Buffer;
```

How is buffer mutex problem solved?
Remote Procedure Call

- Common **operating system service** for client-server model synchronization
  - Implemented with messages
  - Parameter marshalling
    - Semantics remain, implementation may change
  - Mutex problem
    - Combines monitor and synchronized messages?
      - Automatic mutex for service
    - Multiple calls active simultaneously?
      - Mutex problems solved within called service
  - Semantics similar to ordinary procedure call
    - But no global environment (e.g., shared array)
  - Two-way synchronized communication channel
    - Client waits until service completed (usually)
**RPC System Structure**

![RPC System Structure Diagram](image)

**RPC Module**

```plaintext
module mname
  op opname(formals) [returns result] Export public ops
  body
    variable declarations;
    initialization code;
    proc opname(formal identifiers) returns result identifier
      declarations of local variables;
      statements
    end
    local procedures and processes;
  end mname

Call: call mname.opname(arguments)
```
RPC Example: Time Server

```java
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[] on next slide)
```

- Internal process
  - Keeps the time
  - Wakes up delayed clients
- Service RPC’s:
  - time = TimeServer.get_time();
  - TimeServer.delay(10);
### RPC(3)

**NAME**

rpc - library routines for remote procedure calls

**SYNOPSIS AND DESCRIPTION**

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

```c
#include <rpc.h>

char *host;

u_long prognum, versnum, procnum;

char *in, *out;

xdrproc_t inproc, outproc;
```

#### Example

```sh
Linux machine>> man rpc
```

### Remote Method Invocation (RMI)

```java
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

- **Java RPC**
  - Start rmiregistry
    - Stub lookup (default at port 1099)
  - Start rmi server
    - Server runs until explicitly terminated by user

```sh
java -classpath classDir example.hello.Server &
start rmiregistry &
```

### Remote Procedure Call (RPC)

Reference:

- [Remote Method Invocation (RMI)](http://java.sun.com/j2se/1.5.0/docs/guide/rmi/hello/hello-world.html)
package example.hello;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Server implements Hello {
    public Server() {
    }
    public String sayHello() {
        return "Hello, world!";
    }
    public static void main(String[] args) {
        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);
            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

package example.hello;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
    private Client() {
    }
    public static void main(String[] args) {
        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

Output: Server ready
Output: response: Hello, world!
Summary

- Distributed communication with messages
  - Synchronization and communication
  - Computation time + communication time = ?

- Higher level concepts
  - Guarded commands (theoretical background)
  - CSP (idea) & Occam (application)
  - Named Channels (ok without shared memory?)
  - Rendezvous
  - RPC & RMI (Java)