Concurrent Programming (RIO) 30.11.2009

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

Lecture 9: Channels and RPC
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
Synchronization levels (2/5)

Process A

asynchronous?

... 
X = f(..); 
send X to B 
...
X: 10

OS kernel

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

Process B

... 
receive X from A
Y = f(X);
...
X: 5

DC
receive
OS kernel
Synchronization levels (4/5)

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

```
X: 10
```

```
receive X from A
Y = f(X);
```

```
X: 5
```

Synchronization levels (5/5)

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

```
OS kernel
... send
```

```
DC receive
```

```
Y = f(X);
```

```
X: 5
```

```
DC receive
```

```
OS kernel
... receive
```

```
synchronous?
```

30.11.2009 Copyright Teemu Kerola 2008
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 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

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

Greatest common divisor

http://en.wikipedia.org

Predikaatti-muunnos-semantikka

Vartioitu lauseke

Can be also input/output statement
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 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 (e₁, ..., eₙ);
  - Source?port (x₁, ..., xₙ);
- **Binding**
  - Communication with **named processes**
  - Matching types for communication
- **Example**: Copy (West => Copy => East)

<table>
<thead>
<tr>
<th>West:</th>
<th>Copy:</th>
<th>East:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>do true -&gt;</strong></td>
<td><strong>do true -&gt;</strong></td>
<td><strong>do true -&gt;</strong></td>
</tr>
<tr>
<td>Copy!c;</td>
<td>West?c;</td>
<td>Copy?c;</td>
</tr>
<tr>
<td>...</td>
<td>East!c;</td>
<td>...</td>
</tr>
<tr>
<td>od</td>
<td>od</td>
<td>od</td>
</tr>
</tbody>
</table>
OCCAM Language

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

OCCAM Example

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
Inmos Transputer

- B0042
- 2D array
- 10 boards
  420 cpu’s
- 30 boards
  1260 cpu’s

http://www.cs.bris.ac.uk/~dave/transputer.html

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

producen integer x
  loop forever
  p1: x ← produce
  p2: ch ← x

consumer
  integer y
  loop forever
  q1: ch ⇒ y
  q2: consume(y)
```

many readers/writers?
same process writes and reads?

buffer size?
Filtering Problem

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


Filtering Problem with Channels

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>char c, previous ← 0</td>
<td>char c</td>
</tr>
<tr>
<td>integer n ← 0</td>
<td>integer m ← 0</td>
</tr>
<tr>
<td>nC ← previous</td>
<td>loop forever</td>
</tr>
<tr>
<td></td>
<td>no last char?</td>
</tr>
<tr>
<td>p1: nC ← c</td>
<td>q1: pipe ← c</td>
</tr>
<tr>
<td>p2: if (c = previous) and (n &lt; MAX – 1)</td>
<td>q2: outC ← c</td>
</tr>
<tr>
<td>p3: n ← n + 1</td>
<td>q3: m ← m + 1</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>p4: if n &gt; 0</td>
<td>q4: if m &gt;= K</td>
</tr>
<tr>
<td>p5: pipe ← intToChar(n+1)</td>
<td>q5: outC ← newline</td>
</tr>
<tr>
<td>p6: n ← 0</td>
<td>q6: m ← 0</td>
</tr>
<tr>
<td>p7: pipe ← previous</td>
<td>q7:</td>
</tr>
<tr>
<td>p8: previous ← c</td>
<td>q8:</td>
</tr>
</tbody>
</table>
Matrix Multiplication with Channels

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

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

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

How to initialize everything?

How to synchronize everything?

Process Array for Matrix Multiplication

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
Algorithm 8.3: Multiplier process with channels

```
integer FirstElement
channel of integer North, East, South, West
integer Sum, integer SecondElement
loop forever
  p1: North \rightarrow SecondElement
  p2: East \rightarrow Sum
  p3: Sum = Sum + FirstElement \cdot SecondElement
  p4: South \leftarrow SecondElement
  p5: West \leftarrow 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

```
integer FirstElement
channel of integer North, East, South, West
integer Sum, integer SecondElement
loop forever
  either
    p1: North \rightarrow SecondElement
    p2: East \rightarrow Sum
    If message from North available, do this
  or
    p3: East \rightarrow Sum
    p4: North \rightarrow SecondElement
    If message from East available, do this
    p5: South \leftarrow SecondElement
    p6: Sum \leftarrow Sum + FirstElement \cdot SecondElement
    p7: West \leftarrow Sum

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

Discussion 2
Discussion 3
Dining Philosophers with Channels

- Each fork $i$ is a process, 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>for 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>$p_1$: think</td>
<td>$q_1$: forks[i] &lt;- true</td>
</tr>
<tr>
<td>$p_2$: forks[i] =&gt; dummy</td>
<td>$q_2$: forks[i] =&gt; dummy</td>
</tr>
<tr>
<td>$p_3$: forks[i] =&gt; dummy</td>
<td>$q_3$:</td>
</tr>
<tr>
<td>$p_4$: eat</td>
<td>$q_4$:</td>
</tr>
<tr>
<td>$p_5$: forks[i] =&gt; true (would false be ok?)</td>
<td>$q_5$:</td>
</tr>
<tr>
<td>$p_6$: forks[i] &lt;= true</td>
<td>$q_6$:</td>
</tr>
</tbody>
</table>

- Would it be enough to initialize each forks[i] <= true ?
  - Do you really need forks[i] => dummy in fork i? Why?
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 \texttt{service(parm, result)}
  - Server is accepting process \texttt{accept service(p, r)}
  - Server is active process
  - Language construct, no mapping for real system nodes

\begin{algorithm}
\textbf{Algorithm 8.6: Rendezvous}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{client} & \textbf{server} \\
\hline
integer parm, result & integer p, r \\
loop forever & loop forever \\
q1: parm $\leftarrow \ldots$ & q2: accept service(p, r) \\
p1: server.service(parm, result) & q3: \\
p2: use(result) & r $\leftarrow$ do the service(p) \\
\hline
\end{tabular}
\end{center}
\end{algorithm}

- 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

Export public ops defined before task body

task body Buffer is

\( B: \text{Buffer}_{\text{Array}}; \)
\( \text{In}_{\text{Ptr}}, \text{Out}_{\text{Ptr}}, \text{Count}, \text{Index} := 0; \)

\( \ldots \) Buffer.Append (456);
\( \ldots \) Buffer.Append (333);
\( \ldots \)

\( \ldots \) Buffer.Take(x);
\( \ldots \) Buffer.Take(y);

\( \ldots \)

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
          I := B(Out_Ptr);
          end Take;
          Count := Count - 1; Out_Ptr := Out_Ptr + 1;
        or
terminate
    end select;
  end loop;
end Buffer;

Terminates when no rendezvous processes available? Tricky! How to know? No concurrent operations!
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

![Diagram of RPC System Structure]

RPC Module

```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

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
    }

(And00 Fig 8.1)

(process Clock() on next slide)

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);
    }
}

- Internal process
  - Keeps the time
  - Wakes up delayed clients
- Service RPC's:
  - time = TimeServer.get_time();
  - TimeServer.delay(10);
Remote Method Invocation (RMI)

```java
class example.hello.Server {
    public String sayHello() throws RemoteException {
        return "Hello, World!";
    }
}
```

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

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

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();
        }
    }
}

Output: Server ready

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: 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)