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)

- 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 (10)

Usual case

Synchronous?
reliable comm

Synchronous?

Synchronization levels (2/5)

Message Passing

- Symmetric communication
  - Cooperating processes at same level
  - Both know about each other's 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

Guarded command

 guarded statement
in the if

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, and the value of e to B
  - input statement: A?(e)
    - 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)
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 Hatii-2 in AA
  - Automatic message routing for channels
  - Programmed with OCCAM

Inmos Transputer

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

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”


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!

Inmos Transputer

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

Filtering Problem with Channels

- 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”

Matrix Multiplication with Channels

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\times
\begin{pmatrix}
10 & 12 \\
10 & 12 \\
10 & 12
\end{pmatrix}
=
\begin{pmatrix}
42 & 6 \\
105 & 18 \\
168 & 30
\end{pmatrix}
\]

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

27 processes, 24 channels

contains 1 row, sends it down
one element at a time

West-bound multiply-add, South-bound copy North

contains 1 value, makes up three multiply-adds, forwards values down

Matrix Multiplication with Channels

Algorithm 8.3: Multiplier process with channels

integer FirstElement
integer North, East, South, West
integer Sum, integer SecondElement

loop forever
    \(\text{wait 1st for this (*)}
    \)
    \(\text{and then for this}
    \)

p1: North \rightarrow SecondElement
p2: East \rightarrow Sum
p3: Sum \rightarrow Sum + FirstElement \cdot SecondElement
p4: South \rightarrow SecondElement
p5: West \rightarrow Sum

- How to map processes to nodes?
- How to map channels to processes?
  - North channel of one process
  - 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 \(\text{selective input}\)

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

loop forever
    if message from North available, do this
    if message from East available, do this

p1: North \rightarrow SecondElement
p2: East \rightarrow Sum
p3: North \rightarrow SecondElement
p4: South \rightarrow SecondElement
p5: Sum \rightarrow Sum + FirstElement \cdot SecondElement
p6: West \rightarrow Sum

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

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

boolean dummy
loop forever
    \(\text{think}
    \)
    \(\text{forks[0]} \rightarrow \text{dummy}
    \)
    \(\text{forks[0]} \rightarrow \text{dummy}
    \)
    \(\text{eat}
    \)
    \(\text{forks[0]} \rightarrow \text{true}
    \)
    \(\text{forks[0]} \rightarrow \text{dummy}
    \)
    \(\text{mutrex?}
    \)

- Would it be enough to initialize each forks[\(i\)] \(\leq\) true?
  - Do you really need forks[\(i\)] \(\Rightarrow\) 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
  - Server is accepting process
  - Server is active process
  - Language construct, no mapping for real system nodes
  - Asymmetric
    - Can have many similar clients
    - Implementation with messages (e.g.)
      - Service request in one message
      - Arguments must be marshalled
      - 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

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)

Algorithm 9.6: 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 := ...</td>
<td>q1: accept service(p, r)</td>
</tr>
<tr>
<td>p2: server:service(parm, result)</td>
<td></td>
</tr>
<tr>
<td>p3: use(result)</td>
<td>q3: r := 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
RPC System Structure

RPC Module

RPC Example: Time Server

Remote Method Invocation (RMI)

Linux machine>> man rpc

RPC(3)                                                                                     RPC(3)
NAME                                                                                     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.

callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;                     char *host;
u_long prognum, versnum, procnum; u_long prognum, versnum, procnum;
char *inproc, outproc;
decode/encode parameters/results

package example.hello;
import java.rmi.Remote;
import java.rmi.RemoteException;

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

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();
        }
    }
}

Discussion 6

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)