Concurrent Programming (RIO)

Lecture 9: Channels and RPC

Lesson 9

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

Lesson 9

Distributed System

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

Lesson 9

Synchronization levels (10)

Synchronous/asynchronous communication

Discussion 1
Synchronization levels (2/5)

- Process A
  - asynchronous?
  - \(X = f(\ldots);\)
  - send \(X\) to \(B\)
  - \(\text{OS kernel}\)

- Process B
  - \(\ldots\)
  - receive \(X\) from \(A\)
  - \(Y = f(X)\)
  - \(\text{DC receive}\)

Synchronization levels (3/5)

- Process A
  - asynchronous?
  - \(X = f(\ldots);\)
  - send \(X\) to \(B\)
  - \(\text{OS kernel}\)

- Process B
  - \(\ldots\)
  - receive \(X\) from \(A\)
  - \(Y = f(X)\)
  - \(\text{DC receive}\)

Synchronization levels (4/5)

- Process A
  - reliable comm
  - \(X = f(\ldots);\)
  - send \(X\) to \(B\)
  - \(\text{OS kernel}\)

- Process B
  - \(\ldots\)
  - receive \(X\) from \(A\)
  - \(Y = f(X)\)
  - \(\text{DC receive}\)

Synchronization levels (5/5)

- Process A
  - \(X = f(\ldots);\)
  - send \(X\) to \(B\)
  - \(\text{OS kernel}\)

- Process B
  - \(\ldots\)
  - receive \(X\) from \(A\)
  - \(\text{DC receive}\)

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**

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

CSP communication

- **Input/output statements**
  - Destination/port \((e_1, \ldots, e_n)\)
  - Source/port \((x_1, \ldots, x_n)\)
- **Binding**
  - Communication with **named processes**
  - Matching types for communication
- **Example:** Copy (West => Copy => East)
  - **West:**
    - Copy: do true -> Copy:e;
    - …
    - od
  - **Copy:**
    - do true -> West:e;
    - East:e;
    - od
  - **East:**
    - do true -> Copy:e;
    - …
    - od
OCCAM Language

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

OCCAM Example

```c
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
  East ! c1  - - receive next one
```

(O. E. Andrews, p. 331)

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!

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”

Matrix Multiplication with Channels

Matrix multiplication:

\[
\begin{align*}
16 & = 7 \times 2 + 8 \times 2 + 9 \times 0 + 0 \\
30 & = 7 \times 2 + 8 \times 2 + 9 \times 0 + 0
\end{align*}
\]

Process for every multiply-add.

Dining Philosophers with Channels

Each fork i is a process, forks[i] is a channel.

Algorithm 8.5: Dining philosophers with channels

- Would it be enough to initialize each forks[i] <= true?
  - Do you really need forks[i] => dummy in fork i? Why?

Algorithm 8.3: Multiplier process with channels

- How to map processes to nodes?
- How to map channels to processes?
- North-South data flow has priority (*)
  - Waiting even when data-flow East-West available
  - Node on East may be blocked unnecessarily

Discussion 3

Guarded statement

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

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)
**RPC System Structure**

**RPC Module**

- **module name**
  - (p) omname (final) {returns (result)}
  - Export public ops

- **body**
  - variable declarations
  - initialization code
  - (proc) omname (formal identifiers) returns result identifier declarations of local variables
  - statements
  - end
  - local procedures and processes
  - end name

**RPC System Structure**

- Calls
- Qremote
- Qlocal
- P

**RPC Example: Time Server**

- **module TimeServer**
  - op get_time() {returns int} # retrieve time of day
  - op delay(interval) { # delay interval ticks body
    int tod = 0; # time of day
    sem n = 1; # mutual exclusion semaphore
    sem d(n) = (m 0); # private delay semaphores
    queue of (int waittime, int process ID) mapQ
    # when n = 1, tod = waittime for delayed processes
    proc[get time()] {returns time { time = tod; }
    proc[delay(interval)] { assume interval > 0
    int waittime = tod + interval;
    P(n);
    insert(waittime, id) at appropriate place on mapQ;
    V(n);
    P(id(m)); # wait to be awakened
  }

- **process Clock**
  - start hardware timer;
  - while (true) {
    wait for interrupt, then restart hardware timer;
    tod = tod+1;
    P(n);
    while (tod >= smallest wakeup time on mapQ) {
      remove (waittime, id) from mapQ;
      V(id(m)); # awaken process ID
    }
    V(n);
  }

- **end TimeServer**

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

**Remote Method Invocation (RMI)**

- **package example.hello**;
- **import java.rmi.Remote**;
- **import java.rmi.RemoteException**;
- **public interface Hello extends Remote {**
  - **String sayHello() throws RemoteException;**

- **java -classpath classDir example.hello.Server**
- **java -classpath classDir example.hello.Server**

**Linux machine>> man rpc**

- **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.

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

**Discussion 5**

- **RPC**
  - Remote Procedure Call
  - Java RPC
  - RMI
  - Start rmiregistry
  - Start rmi server

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

- **Java -classpath classDir example.hello.Server &**
  - Start java -classpath classDir example.hello.Server
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) {
        Server obj = new Server();
        Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
        // Bind the remote object's stub in the registry
        try {
            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)