Concurrent Programming (RIO)

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

- 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

- Reliable communication
- Synchronous communication
- Asynchronous communication
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

\[
\begin{align*}
  x, y = X, Y & \quad \text{-- statement (unguarded)} \\
  \text{do} & \quad \text{loop command, loop terminates when } x = y \\
  \text{if} & \quad \text{conditional command (itself guarded)} \\
  \text{od} & \quad \text{another statement, also unguarded}
\end{align*}
\]

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 (e₁, …, eₙ) ;
  - Source?port (x₁, …, xₙ) ;
- Binding
  - Communication with named processes
  - Matching types for communication
- Example: Copy (West => Copy => East)

West:

\[
\begin{align*}
  \text{do true} & \rightarrow \\
  \text{Copy} & \rightarrow \\
  \cdots & \\
  \text{od} & \\
\end{align*}
\]

Copy:

\[
\begin{align*}
  \text{do true} & \rightarrow \\
  \text{West} & \rightarrow \\
  \text{East} & \rightarrow \\
  \text{od} & \\
\end{align*}
\]

East:

\[
\begin{align*}
  \text{do true} & \rightarrow \\
  \text{Copy} & \rightarrow \\
  \cdots & \\
  \text{od} & \\
\end{align*}
\]
Concurrent Programming (RIO) 6.2.2012

Lecture 9: Channels and RPC

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
  - 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”? The classic coroutine example

OCCAM Example

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

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 0.1: Producer-consumer (channels)

<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: consum(y)</td>
</tr>
<tr>
<td>buffer size?</td>
<td></td>
</tr>
</tbody>
</table>

Filtering Problem with Channels

Matrix Multiplication with Channels

\[
\begin{bmatrix}
123 \\
456 \\
789
\end{bmatrix}
\times
\begin{bmatrix}
102 \\
012 \\
100
\end{bmatrix}
= 
\begin{bmatrix}
426 \\
105 \\
18
\end{bmatrix}
\]

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

\[
\begin{array}{c}
\text{Result} \\
\text{30} \\
\text{16} \\
\text{0} \\
\text{0}
\end{array}
\]

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, Sum, SecondElement
loop forever
    <-wait of integer North, East, South, West
    <-integer Sum, integer SecondElement

p1: North = SecondElement
    <-wait 1st for this (*)
    <-and then for this
p2: East = Sum
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
  - 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

Relative names?

Discuss

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
        if message from North available, do this
    p2: East = Sum
        if message from East available, do this
    p3: South = SecondElement
    p4: Sum = Sum + FirstElement * SecondElement
    p5: West = Sum
```

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

Discuss

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

```
channel of boolean forks[\(i\)]
philosopher \(i\) / fork \(i\)

boolean dummy
loop forever
    p1: think
    p2: forks[\(i\)] = dummy
    p3: forks[\(i\)] = dummy
    p4: eat
    p5: forks[\(i\)] = true (would false)
    p6: forks[\(i\)] = true (be ok?)

boolean dummy
loop forever
    q1: forks[\(i\)] = true
    q2: forks[\(i\)] = dummy
    q3:
    q4: mutex?
    q5: deadlock
    q6: deadlock
```

- 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
    - 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)
Lecture 9: Channels and RPC

RPC System Structure

RPC Module

RPC Example: Time Server

Remote Method Invocation (RMI)
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)