Concurrence 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

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

OS kernel send DC
```

```
... receive X from A
Y = f(X);
...
X: 10

OS kernel receive DC
```
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

Process A
- reliable comm
- send X to B
  
  X: 10

Process B
- receive X from A
  
  Y = f(X);

- send DC
- receive DC

X: 5

Synchronous?
Synchronization levels (1/5)

Process A

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

X: 10

Process B

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

X: 5

OS kernel
send DC

DC receive OS kernel
Synchronization levels (2/5)

**Process A**

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

```
X: 10
```

**Process B**

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

```
X: 5
```

asynchronous?

Data Com
Synchronization levels (3/5)

---

**Process A**

- \(X = f(\ldots);\)
- \(\text{send } X \text{ to } B\)
- \(X: 10\)

**Process B**

- \(Y = f(X);\)
- \(\text{receive } X \text{ from } A\)
- \(X: 5\)

---

**Asynchronous?**

---

30.11.2009

Copyright Teemu Kerola 2008
Synchronization levels (4/5)

Process A

reliable comm

...
X = f(..);

send X to B

...

X: 10

Process B

...
receive X from A

Y = f(X);

...

X: 5
Synchronization levels (5/5)

Process A

...  
X = f(..);

send X to B

...  
X: 10

OS kernel

send

DC

synchronous?

Process B

...  
receive X from A

Y = f(X);

...  
X: 5

OS kernel

receive

DC
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 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

Usual case
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

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

Copyright Teemu Kerola 2008

16
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 \((e_1, \ldots, e_n)\);  
  – Source?port \((x_1, \ldots, x_n)\);  

• Binding
  – Communication with named processes  
  – Matching types for communication

• Example: \textbf{Copy} (West => Copy => East)

\begin{align*}
\text{West:} & \quad \text{do true ->} \\
& \hspace{1cm} \text{Copy!c; \quad …} \\
& \hspace{1cm} \text{od}  \\
\text{Copy:} & \quad \text{do true ->} \\
& \hspace{1cm} \text{West?c; \quad East!c; \quad od}  \\
\text{East:} & \quad \text{do true ->} \\
& \hspace{1cm} \text{Copy?c; \quad …} \\
& \hspace{1cm} \text{od}
\end{align*}
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

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

30.11.2009
Copyright Teemu Kerola 2008
Inmos Transputer

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

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

30.11.2009 Copyright Teemu Kerola 2008
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)

<table>
<thead>
<tr>
<th>Channel of integer ch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>producer</strong></td>
</tr>
<tr>
<td>integer ( x )</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: ( x \leftarrow \text{produce} )</td>
</tr>
<tr>
<td>p2: ( \text{ch} \leftarrow x )</td>
</tr>
</tbody>
</table>

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

<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><strong>inC ⇒ previous</strong></td>
<td><strong>loop forever</strong></td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td><strong>no last char?</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Algorithm 8.2: Conway’s problem**

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

<table>
<thead>
<tr>
<th>p1:</th>
<th><strong>inC ⇒ c</strong></th>
</tr>
</thead>
</table>
| p2: | if (c = previous) and  
     | (n < MAX - 1) |
| p3: | n ← n + 1    |
|     | else         |
| p4: | if n > 0     |
| p5: | **pipe ← intToChar(n+1)** |
| p6: | n ← 0        |
| p7: | **pipe ← previous** |
| p8: | previous ← c |
| q1: | **pipe ⇒ c** |
| q2: | outC ← c     |
| q3: | m ← m + 1    |
| q4: | if m >= K    |
| q5: | outC ← newline |
| q6: | m ← 0        |
| q7: |              |
| q8: |              |
Matrix Multiplication with Channels

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

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

\[
\begin{array}{ccc}
7*2 + 8*2 + 9*0 + 0 & \Rightarrow & 16 \\
4*2 + 2*2 + 6*0 + 0 & \Rightarrow & 30 \\
7*2 + 8*2 + 9*0 + 0 & \Rightarrow & 16 \\
1*2 + 2*2 + 0*0 + 0 & \Rightarrow & 0
\end{array}
\]

other processes

channels

\[
\begin{array}{cccc}
\text{Result} & = & 7 + & 16 \\
\text{Result} & = & 8 + & 0 \\
\text{Result} & = & 9 + & 0 \\
\text{Zero}
\end{array}
\]
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 three multiply-adds, forwards values down

How to initialize everything?

How to synchronize everything?
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 8.3: Multiplier process with channels

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

loop forever
p1: North ⇒ SecondElement
p2: East ⇒ Sum
p3: Sum ← Sum + FirstElement × SecondElement
p4: South ← SecondElement
p5: West ← Sum
```

Relative names?

wait 1st for this (*)
and then for this
**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
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>Channel of boolean forks[5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>philosopher $i$</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>boolean dummy</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: think</td>
</tr>
<tr>
<td>p2: forks[i] $\Rightarrow$ dummy</td>
</tr>
<tr>
<td>p3: forks[i⊕1] $\Rightarrow$ dummy</td>
</tr>
<tr>
<td>p4: eat</td>
</tr>
<tr>
<td>p5: forks[i] $\leftarrow$ true (would false be ok?)</td>
</tr>
<tr>
<td>p6: forks[i⊕1] $\leftarrow$ true</td>
</tr>
</tbody>
</table>

- 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
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 ← ...</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 ← 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

Export public ops defined before task body

task body Buffer is
  B: Buffer_Array;
  In_Ptr, Out_Ptr, Count: Index := 0;

  begin
    loop
      select
        when Count < Index’Last =>
          accept Append(l: in Integer) do
            B(In_Ptr) := l;
            end Append;
            Count := Count + 1; In_Ptr := In_Ptr + 1;
        or when Count > 0 =>
          accept Take(l: 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;

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

30.11.2009
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

(P calls \rightarrow Q)

Local application

Local response

Local procedure calls

Local stub

RPC mechanism

Local application

Local response

Local procedure calls

Remote procedure call

Qremote

Remote server application

Local response

Local procedure call

Qlocal

Local application

Local response

Local procedure calls

RPC mechanism

Local stub

Remote procedure call

Remote procedure call

(Sta05 Fig 14.12)
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)
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:
  ```java
  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.

callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;
char *in, *out;
xdrproc_t inproc, outproc;
Remote Method Invocation (RMI)

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

```java
package example.hello;

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

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

More information can be found [here](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();
        }
    }
}
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)