Summary of Interprocess Communication and Coordination

Qiong Cheng

Oct. 8 2007
### Process Communication and Coordination

#### Distributed Process Coordination

- **Process**
  - RPC: JavaRMI, CORBA, DCOM, SOAP
- **OS Kernel**
  - Message Passing
    - pipe: Named pipe
    - FIFO: byte-stream buffer
- **Network Protocol Suite**
  - Socket; secure socket

#### Interprocess Communication

- Name and Directory Service

#### Leader Election

- Distributed Mutual Exclusion

- Transaction ACID
Application: Distributed Hotel Room Scheduler

Message: Process 1 reserves/cancels schedule on the i-th day

Process 1: Local calendar

Process 2: Local calendar

Process 3: Local calendar

Process 4: Local calendar

“Shared Calendar”

Internet
Part I: JavaRMI

- An API standard for building distributed Java systems

- Remote Objects
  - Live on server
  - Accessed as if they were local
1. Server Creates Remote Object
2. Server Registers Remote Object
3. Client requests object from Registry
4. Registry returns remote reference (and stub gets created)
5. Client invokes stub method
6. Stub talks to skeleton
7. Skeleton invokes remote object method
**Steps Involved**

- Define a remote interface
- Write an implementation class (Server) for the remote interface
- Compile the java source files
- Generate Stubs and Skeletons (rmic)
- Create the Client files
- Run the registry (start rmiregistry)
- Run the Server
- Run the Client
//hello.interface

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

//helloimpl.java
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends java.rmi.server.UnicastRemoteObject implements Hello{
    public HelloImpl() throws RemoteException{
        super();
    }
    public String sayHello() throws java.rmi.RemoteException{
        return "Distributed";
    }
}

//contd...
public static void main(String[] args) {
    // install a security manager
    System.setSecurityManager(new RMISecurityManager());
    try {
        // create a server object
        HelloImpl h = new HelloImpl();
        // bind server in registry
        Naming.rebind("helloserver", h);
    } catch (Exception e) {
        e.printStackTrace();
    }
} // end class
//client.java
import java.rmi.*;
public class Client{
    public static void main(String args[]){
        try{
            String mesg;
            Hello stub = (Hello)Naming.lookup("helloserver");
            mesg = stub.sayHello();
            System.out.println("hello world - "+mesg);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
Part II: Event Ordering

Communication is not instantaneous; Synchronizing by times is not a reliable mechanism; Distributed system is casual.

Event dynamics and network delays requires event ordering; Event ordering requires logical global clock.

Each process has three types of events:
- Internal Events
- Message Send Events
- Message Receive Events
Happen-before relation of Events

Note: Happen-before relation is defined to be the transitive closure of the processor orderings and the message passing orderings.
Implementing Logical Time

Each process $p_i$ maintains two clocks:

1. Logical **local** clock: that measures its own progress

2. Logical **global** clock: that is a representation of $p_i$’s view of the logical global time.
Two Different Logical Time

1. Scalar Time

2. Vector Time
Algorithm of Lamport’s Scalar Time

For every process $i$
{
    Initially, $C_i = 0$;

    On event $e$,
    If $e$ is the receipt of message $msg$,
    $C_i = \max(C_i, C_{msg})$

    $C_i \text{++;}$
    If $e$ is the sending of message $msg$,
    $C_{msg} = C_i$

}
Scalar Time

p₁

1 → 2 → 3

p₂

1 → 2 → 3 → 4 → 5 → 7

p₃

1 → 5 → 6 → 7

p₄

8 → 9 → 10
$T = n$-dimensional integer vectors
$\mathbf{v}_t \mathbf{i}[i]$: logical local clock of $p_i$
$\mathbf{v}_t \mathbf{i}[j]$: $p_i$'s latest knowledge of logical clock of process $p_j$
Algorithm of Vector Time

For every process $I$
{
    Initially, $VT_i = [0, \ldots, 0]$;

    On event $e$,
    If $e$ is the receipt of message $msg$,
        for $j = 1$ to $M$
            $VT_i[j] = \max(VT_i[j], VT_{msg}[j])$

    $VT_i[i]++$;
    If $e$ is the sending of message $msg$,
        $VT_{msg} = VT_i$;
}

Vector Time

p_1

p_2

p_3
References