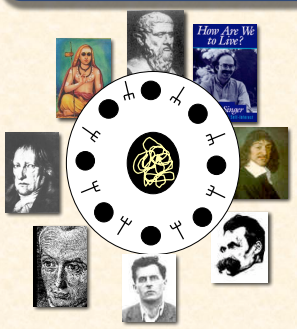
Reintroduction to Concurrency

The execution of a concurrent program consists of multiple processes active at the same time.

Dining philosophers problem



Each philosopher spends some time thinking and some time eating

In order to eat, he needs two forks

When he finishes thinking he picks up the forks on either side of him, eats for a while, then replaces the forks and starts thinking again

How do we ensure that no philosopher starves? What are the properties of the system?

Exercise

- Solve the Dining Philosophers problem!
  - 8 to a group
  - 10 minutes max
  - Devise a strategy that each philosopher can follow blindly (i.e. an algorithm)
  - No philosopher should starve
  - Test your strategy with your group
  - No Solution? What are the problems?
  - A Solution? Show us.

Philosophical Possibilities

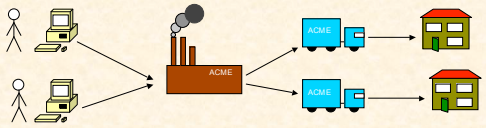
1. Each philosopher simultaneously picks up the fork on his left, then goes for the one on his right, which will be held by another philosopher – deadlock
2. A philosopher can simultaneously pick up the two forks when they become available – atomic actions
3. An important philosopher can take a fork away from a less important one – pre-emptive acquisition

Why do things concurrently?

- **Performance** – execute as fast as possible
- **Throughput** –get as much done as possible
- **Efficiency** – minimise the time spent unproductively
- **Programmability** – some problems are naturally concurrent…what problems?

Logical concurrency

- Several happen together in the designer's mind
  - Accept new orders
  - Dispatch good from the factory



Multitasking

- *Multitasking* is where a single processor executes more than a single activity
  - Shares its times between the activities
  - Periodically the processor changes from one task to another
  - Tasks run "together", rather than one after the other
  - Executes a *context switch* to change its state
  - The order and frequency of context switches are controlled by a *scheduling algorithm*

True concurrency

- True concurrency – sometimes referred to as *parallelism* – happens when a system has more than one processor working on a job
  - Multiprocessor workstation
  - Dedicated parallel computer
  - Co-operating systems on a network

Processes

- Most operating systems have the notion of a *process*
  - An executing program
- Typically, processes are somewhat isolated from each other – each owns
  - An address space
  - A set of resources such as open files, sockets etc.
- By default, processes don't directly interact
  - ✓ No interference
  - × No co-operation

Concurrency in processes

- A system will usually have several processes running
- The processes co-operate to provide a *system*, rather than in providing a single *task*
- Each task is provided by a single "simple" program of the "normal" kind – a sequence of instructions

Threads

- *Threads* (sometimes called *lightweight processes*)
- Lots of executing sequences within a single process
  - Share memory, files, windows, …
  - ✓ Can interact
  - × Can interfere

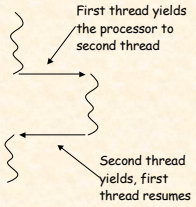
Threads are provided by some operating systems as standard (Win32, Solaris) , and by others as run-time libraries (some Unix variants)

Thread scheduling

- How do we order the execution of threads?
- Are all threads equally important?
  - If yes – give each the same time on the processor
  - If no – give some more time, or run them preferentially
- Can the programmer force a thread to keep running?
  - If yes – rely on the programmer to let other threads have a chance
  - If no – forcibly evict a thread and run another

Co-routines

- Co-routines are a *co-operative* approach to multitasking
  - A thread starts running, and keeps running until it voluntarily gives up the processor



If a thread doesn't yield, it keeps the processor

- ✗ Locks out other threads
- ✓ Lets the programmer keep running something important

More control, more work…

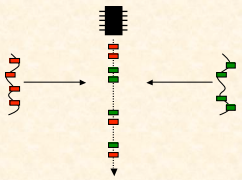
Pre-emptive scheduling

- Simplicity doesn't offer enough compensation for the disadvantages of co-operative scheduling
  - All too easy for a thread not to yield
  - Can lead to some very convoluted programming structures
- Almost all modern systems use *pre-emptive scheduling*, where context switches occur based on the rules of the scheduling algorithm
  - No lock-outs, fairer, more potential problems

You still encounter co-routines in some embedded real-time systems where fine control is too important to lose

Interleaving

- If the processor's being shared, there's the possibility that actions from one task will be mixed-in with actions from another
  - This is known as *interleaving*

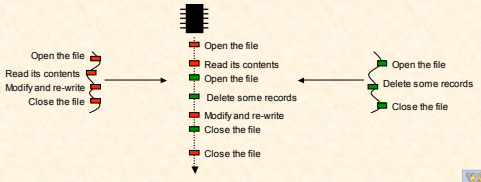


Interleaving

- Interleaving also occurs in truly concurrent systems
- **Any concurrent computation will behave, in the final analysis, like some interleaved sequence of its basic indivisible actions**
- Interleaving of actions is **the** key concept in concurrent programming

Interleaving

- Any kind of interleaving has an important side effect
- Suppose the two tasks share some structure or resource. Interleaving their actions can cause some rather bizarre effects



Interference

- This phenomenon is referred to as *interference* – the actions of two tasks, correct in themselves, conflict when combined concurrently
  - Interference is the basic correctness problem in concurrent programming
- The essence of the problem
  - Tasks use several actions to manipulate the shared structure
  - …and these actions are mixed up together
  - …and the effects of actions in one task conflicts with the behaviour of actions in the other

Interference – inconsistency

- Some actions are naturally "tied together"

people: 21

totalMarks: 1290

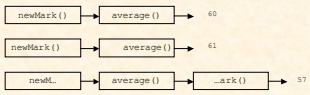
```
newMark()
   add 1 to people
   add total mark
   re-compute average

average()
   get number of people
   get total marks
   divide one by the other
```

Interleaving made the structure *inconsistent* for a time – in which period a calculation took place on bogus data

57

Interference – race conditions

- The is an example of a *race condition*
  - Different results depending on exactly which action runs first



Atomicity

- Each routine is composed of several actions, and those actions must occur indivisibly – *atomically* – for the result to be correct
- This idea of *atomicity* is the key to avoiding undesirable artefacts from concurrency
  - Identify the *critical sections* of code that must execute without any extraneous actions in between them
  - Make these sections atomic

Locking and signalling

- To implement shared data structures without interference, we need to do one of two things
  - *Signal* when a thread is using the structure, so other threads can check whether it's safe to modify it
  - *Lock* the structure so that, if one thread is accessing it, no-one else can

Signalling protocols are still quite co-operative and flexible, as a thread gets to "check and choose" whether to change a shared structure

Locking is safer and more structured, but sometimes a little restrictive

Synchronisation

- Some concurrently executing tasks have a "natural" ordering with respect to each other
- Tasks need to be able to *synchronise* with each other to enforce such orderings

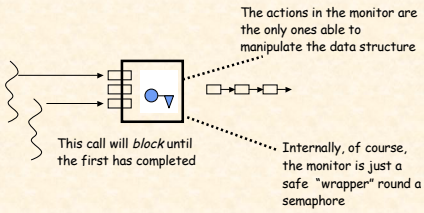
Semaphores

- A semaphore is an abstract data type with two operations, `wait()` and `signal()`
  - `wait()` waits for the semaphore to be dropped, then raises it and continues in one atomic action
  - `signal()` drops the semaphore
- Associate a semaphore with every shared structure and bracket all accesses with `wait()…signal()`

E.W. Dijkstra, *Co-operating sequential processes*, in *Programming languages*, ed. F. Genuys, Prentice-Hall (1968)

Monitors

- Monitors enforce mutual exclusivity on mutually interacting critical sections of code



C.A.R. Hoare, Monitors: an operating system structuring concept, Communications of the ACM **17**(10) (October 1974)

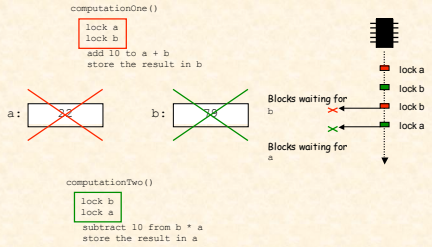
Monitors

- Monitors can provide **language-level** concurrency control
  - The language construct can be checked and analysed
  - Programmers can't avoid the concurrency control, accidentally or maliciously
  - Less error prone when compared to sockets
  - Monitors are the basis of concurrency control in lots of modern programming languages

Monitors and objects

- In object-oriented languages monitors are typically associated with objects
  - Object = data + operations
  - …so if we wrap a monitor around the operations, we protect the data from corruption
- This notion of a *protected value* is very useful
  - Typically speak of "locking a variable" while it's being accessed, and "blocking on a variable" waiting for it to become free

Problems, problems…



Deadlock

- This situation – everyone waiting for everyone else, with no-one making progress – is called *deadlock*
- Well studied problem in computer science
  - Several solutions, but no really general ones
  - *circular waiting*: each task is waiting for (one or more of) the others
  - There are some very convoluted cases – not always just two tasks
- It appears in applications with depressing regularity, and we'll look at it more later

Deadlock

- Deadlock may occur when code has the following properties:
  - Piecemeal acquisition of resources
    - Locks one at a time
  - Resources acquired in any order
    - Circular waiting
  - Ability to hold lock indefinitely

Concept summary

- Activity
  - Processes, threads
  - Actions, interleaving
  - Critical sections, atomicity
  - Synchronisation
  - Signalling and locking
- Constructs
  - Semaphores
  - Blocking
  - Monitors, objects
  - Deadlock

Java Threads

In Java a process is represented by a thread.

To make a thread run you call its start() method.

This registers the thread with the thread scheduler.

start() does not cause the thread to run immediately

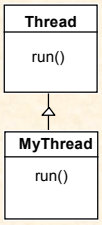
it only makes it eligible to run.

The thread must contend with other threads for the CPU.

When a thread gets to execute, it executes a run() method:

threads in Java

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.



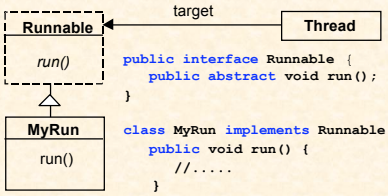
The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

```
class MyThread extends Thread {
    public void run() {
        //......
    }
}
```

```
Thread x = new MyThread();
```


threads in Java

Since Java does not permit multiple inheritance, we often implement the **run()** method in a class not derived from Thread but from the interface Runnable.




```
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}

Thread x = new Thread(new MyRun());
```


threads in Java

Write a Java program that creates three Java threads (by extending `Thread`).

The threads should sleep for a specified amount of time and then print their name and the length of time they slept for to the console.

The three threads should sleep for 500ms, 250ms, 1000ms respectively.

Repeat the exercise implementing `Runnable`

Extend Thread Example

```
class ExtendThread extends Thread{

    private int sleepTime;

    public ExtendThread(String name, int time){
      super(name);
      sleepTime = time;
    }

    public void run(){
      while(true){
          try{
              sleep(sleepTime);
          }catch (InterruptedException e){}
          System.out.println(getName() + " " + sleepTime);
      }
    }
}
```

Extend Thread Example

```
class ExtendMain{
    public static void main(String args[]){
      ExtendThread t1 = new ExtendThread("Thread 1", 500);
      ExtendThread t2 = new ExtendThread("Thread 2", 250);
      ExtendThread t3 = new ExtendThread("Thread 3", 1000);
      t1.start();
      t2.start();
      t3.start();
    }
}
```


Implement Runnable Example

```
class ImplementRunnable implements Runnable{
   private Thread t;
   private int sleepTime;
   public ImplementRunnable(String name, int time){
       sleepTime = time;
       t = new Thread(this, name);
   }
   public void start(){
       t.start();
   }
   public void run(){
       while(true){
         try{
           Thread.sleep(sleepTime);
         }
        catch (InterruptedException e){}
         System.out.println(t.getName() + " " + sleepTime);
       }
   }
}
```


Implement Runnable Example

```
class ImplementMain{
    public static void main(String args[]){
      ImplementRunnable t1 = new ImplementRunnable("Thread 1",
500);
      ImplementRunnable t2 = new ImplementRunnable("Thread 2",
250);
      ImplementRunnable t3 = new ImplementRunnable("Thread 3",
1000);
      t1.start();
      t2.start();
      t3.start();
    }
}
```


Executing a Thread

```
class Interleave {
 public static int c1 = 2;
 public static int c2 = 3;
 public static void main (String[] args) {
  Thread p1 = new P1 ();
  Thread p2 = new P2 ();
  p1.start ();
  p2.start ();
 }
}
```

```
class P1 extends Thread {
 public void run ()  {
  Interleave.c1 = Interleave.c1
  *Interleave .c2;
 }
}
class P2 extends Thread {
 public void run ()  {
   Interleave.c1 = Interleave.c1
                  +Interleave.c2;
 }
}
```

Extend the Thread class and override its run() method.

Executing a Thread

```
class Interleave {
 public static int c1 = 2;
 public static int c2 = 3;
 public static void main (String[] args) {
  Thread p1 = new Thread(new P1());
  Thread p2 = new Thread(new P2());
  p1.start ();
  p2.start ();
 }
}
```

```
class P1 implements Runnable {
 public void run ()  {
  Interleave.c1 = Interleave.c1
                 *Interleave .c2;
 }
}
class P2 implements Runnable {
 public void run ()  {
   Interleave.c1 = Interleave.c1
                  +Interleave.c2;
 }
}
```

Sometimes it is desirable to implement the run() method in a class not derived from Thread but from the interface Runnable.

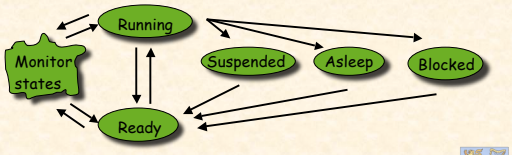
Executing a Thread

When the run() method ends the thread is considered dead.

A dead thread cannot be started again, but it still exists and, like any other object, its methods and data can still be accessed.

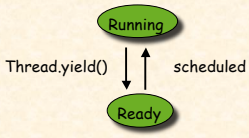
Thread States

When a thread's start() method is called, the thread goes into a ready-to-run state and stays there until the scheduler moves it to the running state. In the course of execution the thread may temporarily give up the CPU and enter some other state.



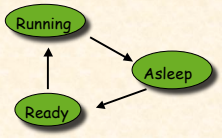
Thread states - Yielding

If there are any other threads in the Ready state, the thread that just yielded may have to wait before it gets to execute again. If there are no waiting threads in the Ready state the thread that just yielded will get to continue executing immediately.



Thread states - Sleeping

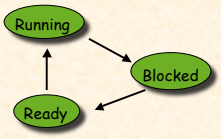
A call to the static sleep() method requests the currently executing thread to cease executing for an approximately specified period in milliseconds.



Note: when the thread finishes sleeping it does not continue execution directly.

Thread states - Blocking

If a method needs to wait for an indeterminable amount of time until some I/O occurrence takes place it should step out of the Running state. This is know as blocking. All Java I/O methods behave this way.



A thread can also become blocked if it fails to acquire the lock for a monitor or if it issues a wait() call. This will be explained later.

Thread Priorities and Scheduling

Every thread has a priority (from 1..10).

All newly created threads have their priority set to that of the creating thread.

Higher priority threads get preference over lower priority threads.

The scheduler generally chooses the the highest-priority waiting thread.

Thread Priorities and Scheduling

If there is more than one waiting thread the scheduler chooses one of them.

There is no guarantee that the one chosen is the one that has been waiting longest.

```
int oldPriority = aThread.getPriority();
int newPriority = Math.min(oldPriority+1,
                   Thread.MAX_PRIORITY);
aThread.setPriority(newPriority);
```

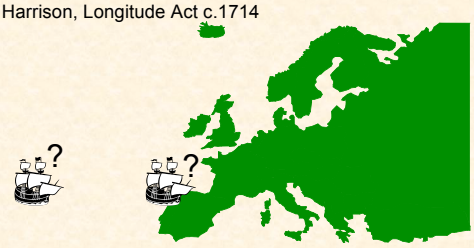
Note: The way thread priorities affect scheduling is platform dependent.

Time and Coordination

Distributed Systems 1

Measuring Longitude

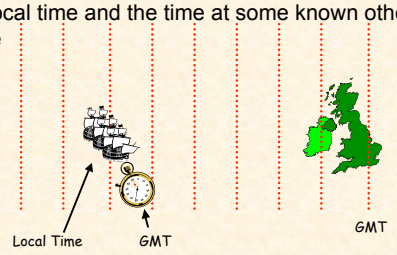
- Harrison, Longitude Act c.1714



- Synchronised clocks used to solve problem of longitude

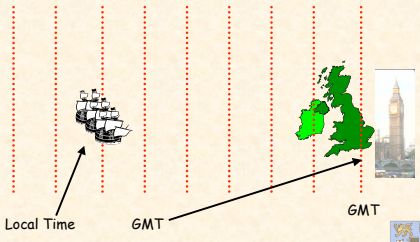
Measuring Longitude

- Position can be calculated provided you know the local time and the time at some known other place



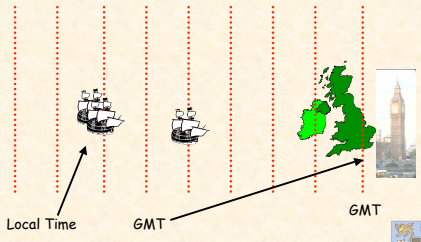
Exercise

- What if there was no portable GMT clock?



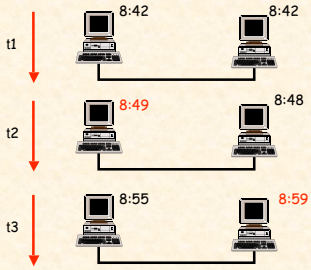
Exercise

- What if there was no portable GMT clock?



Synchronising Computers

-

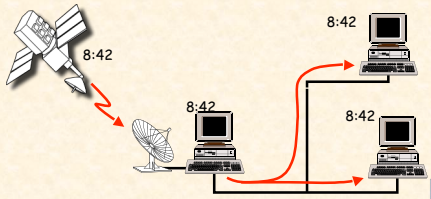


Exercise 1

- How can we agree the time?
  - Clocks "drift"
  - Spend 5 minutes thinking about how you can synchronise your watch to mine
  - We'll spend some time trying to find some solutions, and working out what issues there are to be solved

Time and Co-ordination

- Clock Synchronisation
  - External Synchronisation – refer to external time source

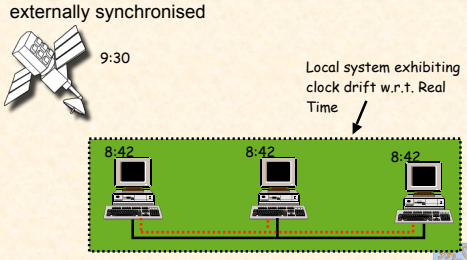


Time and Co-ordination

- Internal Synchronisation – refer to each other

Time and Co-ordination

- Internally Synchronised systems are not necessarily externally synchronised



Synchronising Computers

- Clock Drift
  - Crystal based clocks used in computers are subject to clock drift
  - Count time at different rates
  - Frequency of oscillations varies (e.g. Temperature change)
  - quartz crystal drift: 10^{-6}
    - 1 second in 11.6 days
  - Clocks diverge over time

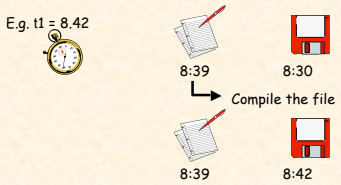
Compensating for clock drift

- If local system time < correct time, the local time can be corrected…
- $$Time_{Local} = Time_{Local} + difference$$
- A few clock ticks will be missed
  12,13,14,20,21,22,…

If local system time > correct time, the local time cannot be corrected immediately because time must always advance!

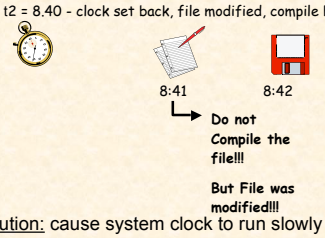
Consider Unix 'make'

- Recompiles only source files that have been modified since the last compilation.
- Compares compiled object file date to source file date.



Consider Unix 'make'

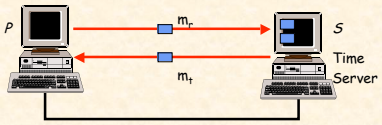
- E.g. t2 = 8.40 - clock set back, file modified, compile begins



Solution: cause system clock to run slowly for a period until it is synchronized with last known correct time

Cristian's method

- Cristian's method for synchronising clocks
  - Central time server process (S) supplies the time according to its own clock upon request.



Cristian's method

- All clients of server S can be synchronised to time

  $T_s \pm$ error

- Note - T_s need not be the correct time.
  - T_s may drift with respect to real time, or it may be synchronised with GPS etc.
- *P* requests time message m_r, and receives time *t* in message m_t.
- Should set clock equal to…

  $t + T_{trans}$

  ...where T_{trans} is time taken to transmit m_t from *S* to *P*
- However, T_{trans} is subject to variation

  i.e. $T_{trans} = t_{min} + t_x$, where $t_x \geq 0$

Cristian's method

- Also, can't measure T_{trans} directly anyway!
- Solution
  - Process *P*, wishing to learn the time from *S*, records the total round-trip time T_{round}.

    $T_{round} = t_2 - t_1$

    t_1 = time m_r is sent
    t_2 = time m_t arrives
  - T_{round} can be measured accurately
    - Round trip time 1- 10 milliseconds on LAN
    - Local clock drift 10^{-5} milliseconds

Cristian's method

- On arrival of m_t at *P*, time *t* in m_t, clock should be set to

  $t + T_{round}/2$

- *Time* $= t + T_{round}/2$ assumes that elapsed time is split equally before and after *S* placed *t* in m_t.
- How accurate is this?

Cristian's method

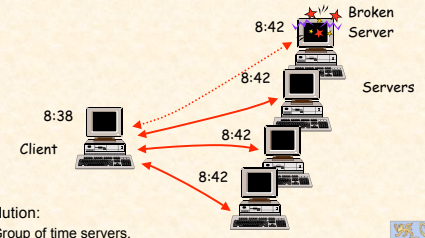
- Determining accuracy of $t + T_{round}/2$
  - Assume t_{min} is known and represents the minimum possible time for message propagation.
  - Earliest point *S* could have placed *t* in m_t was t_{min} after *P* dispatched m_r,
  - Latest point *S* could have placed *t* in m_t was t_{min} before m_t arrived at *P*.
  - By *S*'s clock, the time range for which m_t could arrive at *P* is
    $[t + t_{min}, t + T_{round} - t_{min}]$
  - Width of this range is
    $T_{round} - 2t_{min}$

Cristian's method

- Accuracy of *t* is
  $\pm (T_{round}/2 - t_{min})$
- *P* should set clock to….
  $t + T_{round}/2 \pm (T_{round}/2 - t_{min})$
- To improve accuracy, reduce T_{round}
  - make several requests and take minimum value of T_{round}
- The greater the accuracy required, the smaller the probability of achieving it.

Problems with Cristian's algorithm

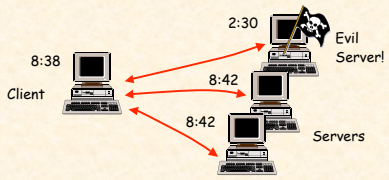
- Single time server is bottleneck
  - If server fails, then cannot synchronise clocks.



- A solution:
  - Group of time servers.
  - Client multicasts requests.

Problems with Cristian's algorithm

- Malfunctioning or impostor time server could wreak havoc!

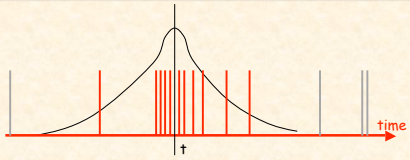


- Solution: Algorithm to detect bad server (E.g. if $t_{bad} < 1/3\ t_{servers}$).
- Solution: Authentication
  - but some security techniques depend on accurate time.

The Berkley Algorithm

- Internal synchronisation algorithm based on Master/Slave model
- Master periodically polls slaves.
- Slaves send back their clock values.
- Master estimates each slaves local clock time by observing round trip times.
  - Similar to Cristian's algorithm.
- Master collects and derives new time *t from* averages values obtained.
  - Includes its own clock.
  - Sets internal clock time to this average.

The Berkley Algorithm



- *t*, on the balance of probabilities, will cancel out the clock drift of each individual clock.
- Unless round trip time of some measurements are very large
  - remove deviant timings
- Fault tolerant average

The Berkley Algorithm

- Master sends back to each slave the amount by which its clock is out of sync with *t*.
  - May be positive or negative value.
- Slaves update their clocks.
  - If slow, update immediately.
  - If fast, slow clock down.
- The clocks of 15 computers typically drift no more than 20 – 25 milliseconds apart.
- System is fault tolerant.
  - If master fails, another can be elected.

Exercise

- Groups of 10
- I am an external time source
- Synchronise to my time
- Take 10-15 minutes to build an approach
- Implement it (i.e. synchronise to my time)

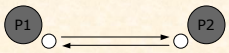
Introduction to Sockets

Remote communication

- Inter-process communication is at the heart of all distributed systems
- Using the network protocol stack on a node is the only way to communicate with remote processes
- Sockets allow to access the network protocol stack
  - Unrelated processes on different nodes can communicate with each other across the network by writing to and reading from sockets

Socket abstraction

- Sockets are a BSD operating system abstraction
- A socket is one endpoint of a two-way communication link between two processes on the network
  - Two sockets on each node are required (client/server model)



Socket abstraction (cont)

- A socket can use different underlying transport protocols
  - TCP, UDP
- A socket is identified
  - By the protocol used
  - By the address of the node
  - By the protocol stack end point (port)
- A process wishing to engage in remote communication must then associate itself with a socket

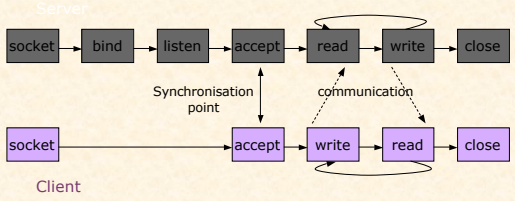
BSD Primitives at server side

- Socket
  - Create a new communication endpoint for a specific transport protocol
- Bind
  - Attach a local address to the newly created socket (e.g., IP address and port number)
- Listen
  - Announce willingness to accept connections
- Accept
  - Block caller until a connection request arrives
  - Create a new socket with the same properties as the original one and returns it to the caller
  - Wait for another connection request on the original socket
- Close
  - Release the connection

BSD Primitives at client side

- Socket
  - Create a new communication endpoint for a specific transport protocol, but bind is not necessary
- Connect
  - Actively attempt to establish a connection, with the transport-level address
  - Block the caller until a connection has been set up successfully
- Send
  - Send some data over the connection
- Receive
  - Receive some data over the connection
- Close
  - Release the connection

Communication using sockets



UDP/TCP protocols

- UDP: unreliable datagram protocol
- TCP: transport communication protocol (stream)
- Datagrams may be lost or received out of sequence while communication over a stream is reliable
- Stream sockets must be connected, datagram need not

Sockets in Java

- Package java.net
- IP level
  - InetAddress
- UDP level
  - DatagramSocket, MultiCastSocket
- TCP level
  - Socket (client side), ServerSocket (server side)
- HTTP level
  - URL, URLConnection, HttpURLConnection

Socket programming with TCP

- Example client-server application
  - Client reads line from standard input, sends to server via socket
  - Server reads line from socket
  - Server converts line to uppercase, sends back to client
  - Client reads, prints modified line from socket

TCP client

```
import java.io.*;
import java.net.*;
class TCPClient{
 public static void main(String argv[]) throws
   Exception {
  String sentence, modifiedSentence;

  BufferedReader inFromUser = new
   BufferReader(new InputStreamReader(System.in));

  Socket clientSocket = new Socket("hostname",
   6789);

  DataOutputStream outToServer = new
   DataOutputStream(clientSocket.getOutputStream());
```

Create input stream

Create client socket

Create output stream attached to socket

TCP client (cont)

```
BufferedReader inFromServer = new BufferReader(new
 InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readline();
outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);
inFromServer.close();
outToServer.close();
inFromUser.close();
clientSocket.close();
}
}
```

Create input stream attached to socket

Send line to server

Read line from server

Close streams and socket

TCP server

```
import java.io.*;
import java.net.*;
class TCPServer{

 public static void main(String argv[]) throws Exception {

  String clientSentence, capitalisedSentence;

  ServerSocket welcomeSocket = new ServerSocket(6789);

  while(true) {
       Socket conSocket = welcomeSocket.accept();

  BufferedReader inFromClient = new
                    BufferedReader(new
                    InputStreamReader(
                    conSocket.getInputStream()));
```

Create welcoming socket at port 6789

Wait for contact by client

Create input stream attached to socket

TCP server (cont)

```
DataOutputStream outToClient = new
 DataOutputStream(conSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalisedSentence = clientSentence.toUpperCase()
 + '\n';

outToClient.writeBytes(capitalisedSentence);

}
}
}
```

Create output stream attached to socket

Read in line from socket

Write out line to socket

End of loop, wait for another client connection

UDP Socket programming

- UDP: no connection between client and server
- Sender explicitly attaches IP address and port of destination to each packet
- Server must extract IP address, port of sender from received packet to return uppercased sentence to sender

UDP client

```
import java.io.*;
import java.net.*;
class UDPClient{
 public static void main(String argv[]) throws Exception{
   BufferedReader inFromUser = new BufferReader(new
  InputStreamReader(System.in));

   DatagramSocket clientSocket = new DatagramSocket();

   InetAddress IPAddress =
  InetAddress.getByName("hostname");

   byte[] sendData = new byte[1024];
   byte[] receiveData = new byte[1024];

   String sentence = inFromUser.readLine();
   sendData = sentence.getBytes();
```

Create input stream

Create client socket without port and IP address

Translate hostname to IP address using DNS

UDP client (cont)

Create datagram

Send datagram to server

Read datagram from server

```
DatagramPacket sendPacket = new
 DatagramPacket(sendData, sendData.length(), IPAddress,
 9876);

clientSocket.send(sendPacket);

DatagramPacket receivePacket = new
 DatagramPacket(receiveData, receiveData.length());
clientSocket.receive(receivePacket);

String modifiedSentence = new
 String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
inFromUser.close();
clientSocket.close();
}
}
```


UDP server

Create datagram socket

Create space for received datagram

Receive datagram

```
import java.io.*;
import java.net.*;
class UDPServer{
  public static void main(String argv[]) throws Exception{
    DatagramSocket serverSocket = new
  DatagramSocket(9876);

    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];

    while(true) {
      DatagramPacket receivePacket = new
  DatagramPacket(receiveData, receiveData.length());
      serverSocket.receive(receivePacket);
```


UDP server (cont)

Get IP address and port

Create datagram to send to client

Write out datagram to socket

```
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalisedSentence = sentence.toUpperCase();
sendData = capitalisedSentence.getBytes();
DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length(), IPAddress, port);

serverSocket.send(sendPacket);
}
}
}
```

End of loop, wait for another datagram

Summary so far

- Socket = abstraction allowing users to write programs that use the underlying communication protocols
- Client/server model

Reading list

- "*Lesson: All About Sockets*", http://java.sun.com/docs/books/tutorial/networking/sockets/index.html
- "*Sockets programming in Java: A tutorial*", http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html

Tutorial

- Design a Chat Room System
- Use any architecture you wish
  - e.g. centralised or decentralised model or …
- Provide Simple UML model
  - I.e. What components in system
  - How do they interact
- Describe technologies you would use
- Justify technology choices

Distributed Systems

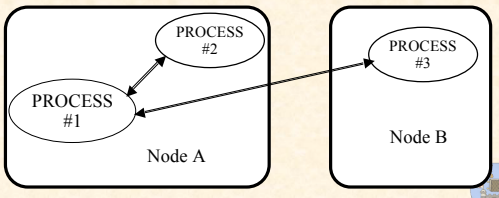
- **Why are distributed algorithms hard?**
- **Complex Problems:** algorithms that are worth distributing, or require distribution, are usually inherently complex
- **Sense data limitations:** distributed algorithms must deal with imperfect information!
  - The propagation delay in transmitting data from one node to another means that the information can be out of date by the time it is used to make a decision.
  - In general in a loosely coupled distributed system, no single process has a coherent view of the entire system at any one time, or can know the state of any other node with 100% confidence

Distributed Systems

- **Failure:** "A distributed system is one where the failure of some node you have never heard of causes your program to crash" (Lamport)
- **Parallelism/concurrency:** Multiple processing elements means that things can go on in parallel
- **Performance and scale:** The algorithms must be efficient and not require change as the system scales
- **Lack of Software Support:** developing distributed applications with technologies such as sockets is very hard. Do technologies such as RMI, CORBA or DCOM help?

Distributed Programming Goal

- Provide abstractions that make it easy to program systems in which processes (potentially on different machines) communicate with each other.



TRANSPARENCY

- **Transparency - The Holy Grail**
- Transparency is the concealment of the underlying physical separation from end users and application programmers.
  - E.g. NFS *allows* files on remote machines to be accessed *as if* they were local.
- The key to achieving ease of programming is the degree to which the system provides transparency

Types of Transparency

- **Access** - local and remote resources accessed identically.
- **Location** - resources can be accessed without knowledge of their location.
- **Concurrency** - several users or application programs can concurrently use shared data without mutual interference.
- **Replication** - multiple copies of resources can be used to increase reliability and performance without users or applications being aware of the replicas.

Types of Transparency

- **Failure** - concealment of faults, allowing users and applications to complete their work despite the presence of faults.
- **Execution** - allows the movement of processes within a system (e.g. so as to load share) without affecting operation of users or applications.
- **Scaling** - allows the system to grow without changing the overall system, structure or application programs.
- **Performance** - none of the list above count for anything if the system does not perform "well".

Paradigm Development

- **Paradigm Development**
- The development of **inter process communication** paradigms has been in an upward direction with each new paradigm presenting a higher level of abstraction to the programmer.
- Raw Network I/O … Protocol Stack … File I/O …. Messages ... Remote Procedure Call ... Distributed Object Invocation ….

Sockets (Stream)

- No boundaries in the data
- stream.
- No end to end type checking.
- Manual conversion and buffering
- of data.
- No transparency in error handling.
- Complex Binding.
- Finished up writing pseudo
- message passing primitives.
  - `Send(ProcessID, Structure)`
  - `Receive(SourceID, Structure)`

Message Passing

- Provide a higher level of abstraction.
- Processes communicate (across a network) by exchanging **messages** where a message is a (variable) sequence of bytes.
- The sender must specify the identity of the intended recipient.
- The main message passing primitives are **`SEND(Dest, Msg)`** and **`RECEIVE(Source, Msg)`** with numerous variations.
  - **N.B. pointers are invalid on the remote node.**

Remote Procedure Call - RPC

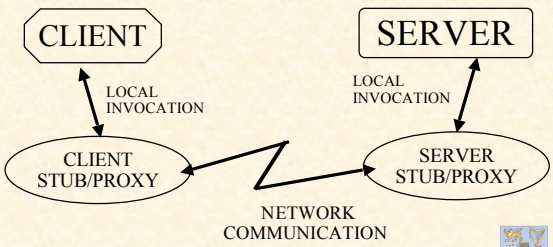
- RPC attempts to make the **procedure call abstraction** available to the distributed systems programmer.
- Appear like local procedure calls at the application level.
  - Stub routines *marshal* the parameters
  - pass them to the runtime for transmission across the network.
- Ideally invoking a remote procedure is identical to calling one in the same address space.
  - This is not totally feasible in practice.
- RPC layered on send/receive/reply.

Marshalling

- *"Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message"*
- Convert into the *on the wire format*.
- Use a common external data representation format to handle heterogeneity issues
  - ordering, character code, etc. NOP if system is homogeneous.
- Marshalling Code is needed.

Paradigm Development

- **Marshalling a remote procedure call**

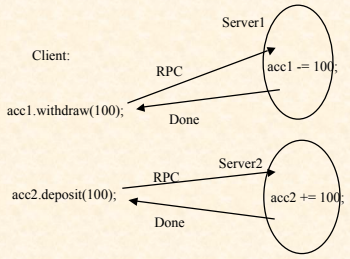


Marshalling

- Define procedures being exported in a separate **Interface Definition Language.**
- Marshalling code (called stub routines) can be automatically produced from this definition for the target language by a compiler/pre-processor.
- This frees the programmer from much of the complexity of distributed systems programming.
- However...

Failure in Remote Procedure Call

- consider moving money between two bank accounts managed by different servers.



Failure in Remote Procedure Calls

- no reply to first request is received?
  - request lost? reply lost? server crashed?
  - has the withdrawal been made or not?
  - retry? but don't want to withdraw money twice!
- Server1 crashes after replying to client?
  - will the client know?
  - will the update to acc1 persist?
- no reply to second request received?
  - as above but what to do about Server1?
  - inconsistency may arise due to partial failure!

Distributed Application programmer must be prepared to handle additional failure modes

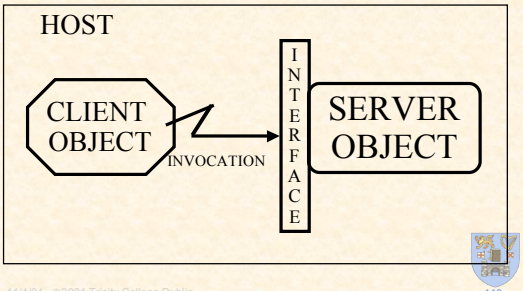
Some programming models returns these failures as language level "events"

Paradigm Development

- **Distributed Object Oriented Programming**
- RPC attempts to make the **procedure call abstraction** available to the distributed systems programmer.
- Object Oriented Distributed Technologies attempt to provide remote access to **objects.**
- *i.e. Make the method call available to the distributed programmer*

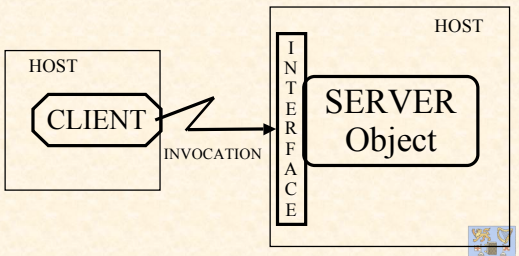
Paradigm Development

- **Object invocation**

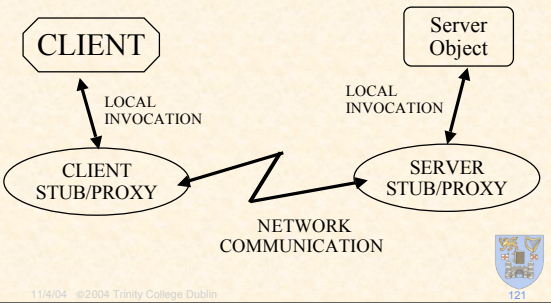


Paradigm Development

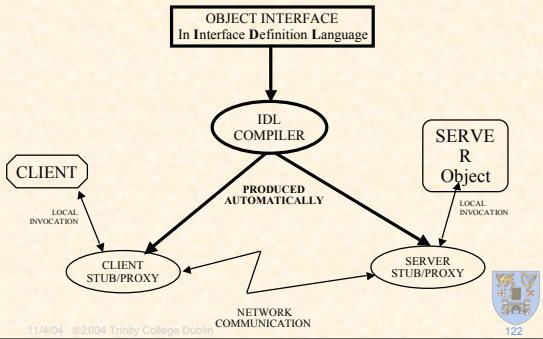
- **Transparent remote object invocation**



Marshalling a remote object invocation

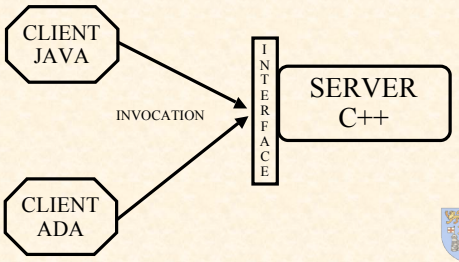


Automatic generation of stubs

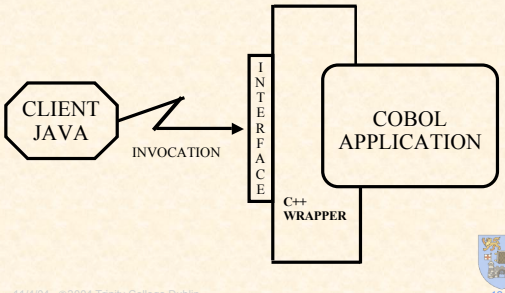


Inter language object invocation

- Interface Description can be Language Independent

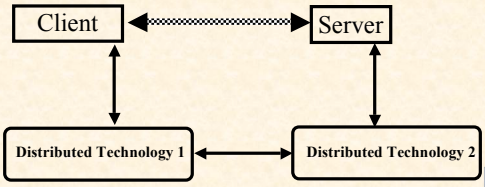


Legacy system integration



Interoperability

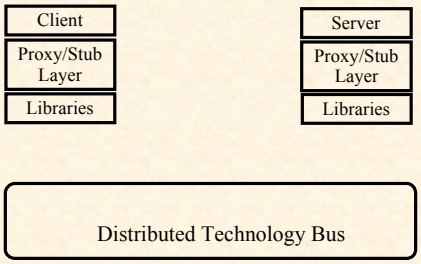
- Enable distributed technologies from different vendors to interoperate.



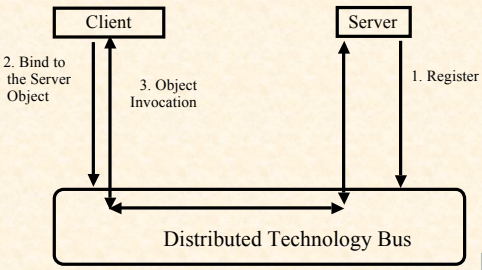
Paradigm Development

- ***Today's applications are tomorrow's components!***
- Define the interface to the application when it is being written so that it can be reused in another application later on.

Architecture



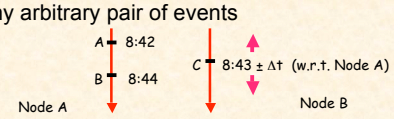
Architecture in Action



Logical Time

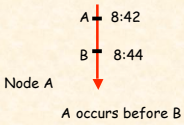
Logical Time

- We wish to order events in a distributed system
- We cannot **perfectly** synchronise clocks across a distributed system
- In general, we **cannot** use physical time to order any arbitrary pair of events



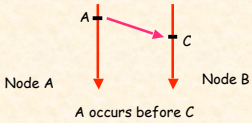
Logical Time

- However, two observations are useful
  - If two events occurred at the same process, then they occurred in the order in which they are observed



Logical Time

- Whenever a message is sent between two processes, the event of sending the message occurs before the event of receiving the message!



- This is the basis of Lamport's 'happened-before' relationship
  - causal ordering
  - Potential causal ordering

Causal Ordering

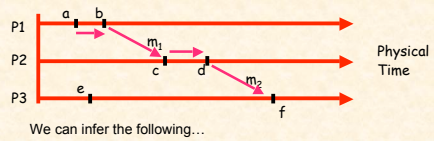
$x \xrightarrow{p} y$ events x and y occurred at process p in order x,y

$x \rightarrow y$ event x 'happened-before' y

$x \rightarrow y$ defined:

- Rule1: if ∃ process p: $x \xrightarrow{p} y$, then $x \rightarrow y$
- Rule 2: For any message m, send(m) → rcv(m)
  where send(m) is the event of sending the message and rcv(m) is the event of receiving the message
- Rule 3: If x,y and z are events such that $x \rightarrow y$, $y \rightarrow z$ then $x \rightarrow z$

Causal Ordering



We can infer the following…

a→b	Rule 1	b→c	Rule 2
c→d		d→f	
a→f	Rule 3		
b→d			
etc.			

Rule 3 allows us to define causal orderings across processes!

Causal Ordering

- Notes:
  - $a \not\rightarrow e$ and $e \not\rightarrow a$ because we cannot determine an ordering for a and e as there is no chain of messages relating them
  - a || e
  - Events a,b,c, .. F may or may not be causally related
  - Our scheme only captures potential causality

Logical Clocks

- Simple mechanism which captures causal ordering via numerical values
- Each process P keeps its own logical clock, Cp, which it uses to timestamp events
- Processes update their logical clocks and transmit the values of clocks as follows
  - Cp incremented before each event is issues at process p
    Cp := Cp + 1

Logical Clocks

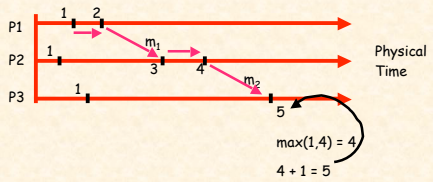
- When process P sends a message m, it piggybacks the value t = Cp on m

  On receiving (m,t), a process q computes Cq := max(Cq,t) and then increments clock before timestamping the event rcv(m)

  i.e. $t_{rcv(m)} = Cq + 1$

- $a \rightarrow b \Rightarrow C(a) < C(b)$
  - where C(x) is timestamp at event x

Logical Clocks



Introduction to RMI

RMI Introduction

- Java Remote Method Invocation (RMI)

- **Sources:**
- `http://java.sun.com/products/jdk/rmi/index.html`
  - for JDK 1.1; includes Java RMI tutorial and specification
- `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html`
  - describes RMI enhancements for upcoming JDK 1.2

RMI Introduction

- Java's Remote Method Invocation (RMI)
  - writing distributed applications in Java (only)
- RMI provides the ability to:
  - define types whose instances may be remote
    - i.e., execute on another Java virtual machine
    - typically another address space or node
  - invoke remote objects transparently
  - pass objects by reference or by value as parameters in a remote invocation
  - obtain a reference to a remote object from a simple name service
  - download the code necessary to access a remote object automatically

RMI basics

- Java is a *class-based* object-oriented programming language
  - Supports *single inheritance*
- A Java *interface* is essentially an abstract class
  - a collection of method *definitions* (and constant declarations)
  - methods and constants declared in an interface are implicitly `public`
- An interface can be derived from *one or more* interfaces
- A class can *implement* one or more interfaces
  - as well as being derived from at most one other class
- The interface and class hierarchies are orthogonal

Defining a remote interface

- A type whose instances may be invoked remotely is defined as a *remote interface*
- A remote interface is simply an interface that extends the interface `java.rmi.Remote` (no IDL!!!)
- The methods of a remote interface must be defined to throw the exception `java.rmi.RemoteException`
- Otherwise syntactically *identical* to any other Java interface

Defining a remote interface

- For example, the interface exported by our game server:

Remote interfaces must be `public`

Import package `java.rmi`

```
package DistributedGame;

import java.rmi.*;

/* Interface exported by the game server. */
/* Allows a player to join in a game. */
public interface GameServer extends Remote {
  Game join() throws RemoteException;
}
```

`join` takes no parameters and returns a `Game`

Making a remote object available

- Remote objects are instances of classes that implement remote interfaces
- Such a class simply implements the methods defined in the remote interface
- To be accessible remotely, a remote object must be *exported*
  - i.e., listen for incoming invocation requests
- An object is implicitly exported if its class derives from `java.rmi.server.UnicastRemoteObject`
  - not necessary; an object may also be exported explicitly (see later)

Making a remote object available

For example, an (outline) implementation of the game server interface:

```
package DistributedGame;

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

public class GameServerImpl
  extends UnicastRemoteObject
  implements GameServer
{
  public GameServerImpl() throws RemoteException {
   super();
   ...
  }

  public Game join() throws RemoteException {
   ...
  }
```

UnicastRemoteObject constructor exports the object

Making a remote object available

- Remote objects are typically hosted by server programs
- A server program is simply a Java application that creates one or more remote objects
- Such a server must install an appropriate security manager - typically an instance of `java.rmi.RMISecurityManager`
- A server may advertise references to one or more of the objects that it hosts via the local RMI registry
- The registry allows a binding between a URL and an object reference to be made and subsequently queried by potential clients

Making a remote object available

```
/* the server mainline */
public static void main(String args[]) {

  System.setSecurityManager(new RMISecurityManager());

  try {
   GameServerImpl server = new GameServerImpl();
   Naming.rebind("//mars.dsg.cs.tcd.ie/GameServer",
server);
    System.out.println("GameServer installed.");
  } catch (Exception e) {
   System.out.println("GameServer exception.");
  }
  /* now the server object is waiting for requests */
 }
}
```

create and install a security manager

create a server object and register it with the registry

Accessing a remote object

- From a client's perspective access to a remote object is transparent...
- ...except that the client has to be prepared to handle `java.rmi.RemoteException` when invoking a remote object
- If `java.rmi.RemoteException` is raised the caller has no guarantee as to how far the invocation proceeded before failing
- The client will normally obtain a reference to its first remote object from the registry

Accessing a remote object

- For example, an implementation of an applet that wants to join a game:

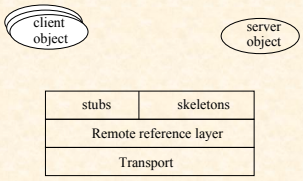
```
package DistributedGame;

. . .
import java.rmi.*;

public class TicTacToe extends Applet  implements . . . {

public void init() {
 . . .
 try {
  server = (GameServer)Naming.lookup("//"
            + getCodeBase().getHost() + "/GameServer");
  game = server.join();
 } catch (RemoteException e) {
  showStatus("Applet initialisation failed.");
  stop();
 }
}
```

Note cast to remote *interface* type

Invocation of remote object is syntactically identical to any other invocation

RMI details

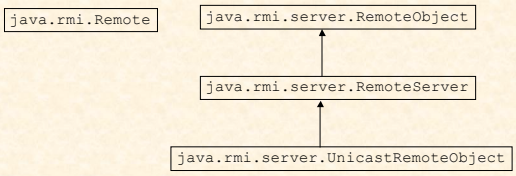
- **RMI Architecture 1**
- In principle, a three layered architecture:



RMI details

- Main class hierarchies:

Implements remote reference layer
- semantics of remote objects (`equals`)
Only implemented for singleton objects



RMI details

- A class can implement several remote interfaces as well as implementing methods that do not belong to any remote interface
  - such methods cannot be invoked on a remote object
- Clients always refer to remote objects via their remote interface types not their implementation class type
- A reference to a remote object can be passed as a parameter or returned as a result of any method invocation
- In particular, remote objects are passed *by reference* in remote method invocations
- **When passing a remote object as a parameter, the stub for the remote object is passed**

Concurrency control

- For example,
- Given two references, `r1` and `r2`, to a remote object (transmitted in different invocations):
  - `r1 == r2` is `false!!!!` (different stubs)
  - `r1.equals(r2)` is `true` (stubs for the same remote object)

RMI details

- In fact, the parameters to or return value from a remote method can be of any Java type that is *serialisable*
  - i.e., non-remote objects can be passed and returned
- Non-remote objects are passed by value in remote method invocations
  - i.e., a copy of the object is passed
- **Effectively, a new object is created for each invocation**

RMI details

- For example,
- Given variables, `l1` and `l2`, referring to the same non-remote object transmitted in different invocations:
  - `l1 == l2` is `false` (different copies)
  - `l1.equals(l2)` is `true/false`
  - (depending on whether the state of the source object changed between the invocations)

Example 2

- The original version of our TicTacToe game has the players poll the game to determine when it is their go
- An alternative is to have the game make a callback to the player whenever the opponent makes their move
- Now player must also be implemented as remote objects (capable of being invoked remotely)
- We will need the following new interfaces:

Example 2

```
/* Interface exported by the game. */
/* Allows a player to particpate in a game with another
player. */
public interface Game extends Remote {
  /* return a handle for the game */
  String getName() throws RemoteException;
  /* determine if caller has first move */
  boolean firstMove(Player player) throws RemoteException;
  /* handle a move by the caller */
  void reportMove(Player player, Point move) throws
RemoteException;
}

/* Callback interface exported by each player. */
public interface Player extends Remote {
  /* handle a move by the remote opponent */
  void reportMove(Point move) throws RemoteException;
}
```


Example 2

- The main implementation problem is that the client applet can't extend `UnicastRemoteObject` as before
- The client must take responsibility for exporting itself explicitly

Example 2

```
package DistributedGame;

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

public class TicTacToe extends Applet implements Player,... {

public void init() {
 . . .
 try {
  UnicastRemoteObject.exportObject(this);
  server = (GameServer)Naming.lookup("//"
            + getCodeBase().getHost() + "/GameServer");
  game = server.join();
 } catch (RemoteException e) {
  showStatus("Applet initialisation failed.");
  stop();
 }
}
. . .
```


Course Project

Tutorial

- Outline the design of an online travel booking service
- Support creation and booking of full itineraries
  - Flights
  - Hotels
  - Shows
  - Etc.
- Discuss how this might be constructed using technologies you know about
- Discuss problems you might encounter