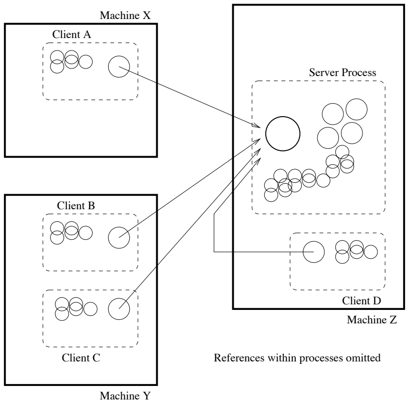
Remote Procedure Call — Java RMI

- Dr Peter Dickman
- Email: pd@dcs.gla.ac.uk
- Materials: `/users/students4/software/public/GCM`

- High-speed summary of the DAS4 lectures on programming with RMI:
  - RPC is a key building block for distributed systems
  - Higher level than socket programming
  - Learning objectives:
    - Understand what is happening “under the hood”
    - Be able to use these technologies
    - Be able to explain what is happening and why
- Two pieces of practical work:
  - Completely trivial warm-up exercise — issued today, complete asap
  - Simple test of use of RMI — out Friday 20th, back Friday 27th Jan

Extending the Reference/Invocation

- Basic concepts for OO programmers:
  references to objects; invocations of object methods
- More generally: procedures/functions are invocable code fragments
  encapsulate state with related code fragments for manipulating it



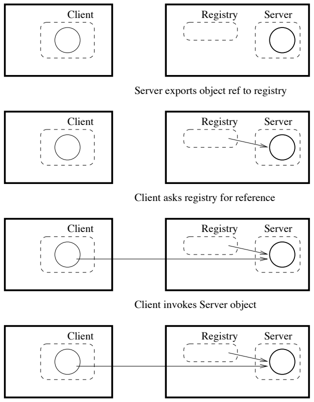
References within processes omitted

- Why assume that references are forced to stay within the process boundary?
- Why restrict invocations to be within the callers process?

Acquiring Remote References

- Can acquire a Remote Ref as an invocation parameter or result
- But there’s a bootstrapping problem. . .

- Alternative approach:
  - Expose/acquire via reference server
  - Another bootstrap problem?
  - “magic” libraries fix this
  - Name servers match names to refs
  - Java RMI has the `rmiregistry`



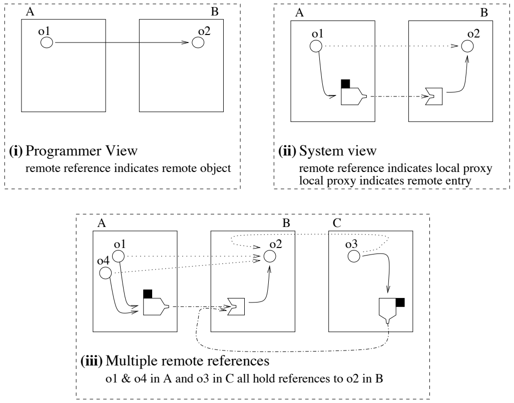
Server exports object ref to registry

Client asks registry for reference

Client invokes Server object

Implementing Remote References

- Generate an illusion of “remote” references
- Utilise local references to hidden objects that exploit sockets etc
- Generate underlying code, utilise network libraries, extended run-time



(i) Programmer View
remote reference indicates remote object

(ii) System view
remote reference indicates local proxy
local proxy indicates remote entry

(iii) Multiple remote references
o1 & o4 in A and o3 in C all hold references to o2 in B

What happens during an RPC/RMI call?

- Invocation is to a local stub object, providing same interface
- It marshals/serializes/flattens the arguments, passes into network

- On receipt at remote process, call and args are unpacked
- A thread, and associated stack, is created/acquired and invoked
- New, remote thread invokes the remote object

- Results are returned by reversing these actions

The effect on the stack frames:

Concurrency Implications

- Multiple incoming calls create/acquire multiple threads

- Creating threads vs Thread pools

- Is the concurrency significant?
- Is it bounded? If so, how?
- Could the server be overloaded?

- Size the system: number of calls ∗ duration

- Dynamically restricting the amount of concurrency?

Parameter Passing in an RPC

- Arguments are at the caller/client, but needed by the callee/server
- Options:
    - server makes RPCs back to argument object
        - but how many calls, is this efficient?
    - argument object is migrated to server for duration of call
        - do others get to access it? if so local vs remote issues
    - argument object migrated to server forever
    - argument object is copied to server
        - but now two copies, are they kept consistent?
            - if so how? If not, what happens?
        - Is one copy discarded after call completes?
            - if so, which one?
        - If copying occurs, how deep is the copy?

Java Parameter Passing

- normal invocations
  - built-in values
    - passed by copy/value to the relevant parameter/register
  - normal Java objects
    - passed by reference, i.e. pointer to object is passed by copy/value
- RMI calls:
  - built-in values
    - passed by copy/value to the relevant parameter/register
    - machine heterogeneity: big/little endian, width of integers etc
    - ensure we have the same value, not the same bit-pattern
  - remotely invocable objects
    - passed by reference
    - a remote reference is constructed at the callee side of the call
    - regardless of whether the object was local or remote at the caller
  - normal java objects
    - interesting and awkward question. . .

Passing local objects in remote calls

- Could forbid this. But very restrictive.
- Could make every Java object remotely invocable. Too expensive.
- Could dynamically make objects invocable. Horrible security implications.
- Could permanently migrate the object. Renders it unusable locally.
- Could temporarily migrate the object. Blocks other calls. Deadlocks?
- Also, if the object contains references, do we migrate them too?

- Solution is to deep copy the object: copy it and everything it references
- View the copy as separate, no attempt to maintain consistency

RMI semantics:

- normal java objects are passed by deep copy/value
- built-in values are passed by copy/value
- remotely invocable objects are passed by reference

Call Semantics

- Normal invocation is exactly-once
- RPC/RMI does not give exactly-once semantics

- A call may fail *before*, *during* or *after* execution at remote site
- Simply repeating a call that doesn't reply may give multiple execution

- Idempotent calls are very helpful: can repeat them safely
  - adding a value into a variable is not idempotent
  - assigning a value into a variable is idempotent
    - in the absence of parallel confounding activity

Definition: a function f is *idempotent* if and only if $\forall x : f(f(x)) = f(x)$

Remote Exceptions in Java RMI

- because of the possibility of problems (e.g. no server present)
- all remotely invocable methods potentially throw a RemoteException
- these are generated automagically by the run time support

- because the stub objects have to be generated, it's important to indicate which methods are remotely invocable; they form a remote interface
- because the stubs/remote refs may throw remote exceptions, it's important to be aware of them as different and provide try-catch clauses

- Overall effect:
  - remotely invocable objects and remote invocations do not look exactly like normal local ones, but they are very similar
  - remote references do look like local references; until you use them

Remote Interfaces

- If instances of a class are supposed to be remotely invocable:
  - The class must extend `UnicastRemoteObject`
  - The class must implement an interface that describes the methods it makes available to holders of remote references to it.
  - Such interfaces must extend `Remote`
  - the methods must declare they can throw `RemoteException`
    - even though their implementations will *not* do this explicitly
- Remote method parameters and results must be acceptable
  - built-in types are acceptable
  - references to remotely invocable objects are acceptable
  - references to normal Java objects are only acceptable if the object is an instance of a class which implements Serializable
    - which is a special interface, requiring no specific methods
- References to a remote object indicate the hidden stub
- Their type is the remote interface type, not the class

Inheritance and Java RMI

- Interfaces use multiple inheritance
  - This means the remote interfaces form a DAG (actually a semi-lattice) in the inheritance hierarchy, descended from `Remote`
- Classes use single inheritance
  - This means the remotely invocable classes form a tree in the inheritance hierarchy, descended from `UnicastRemoteObject`
- Inheriting from `UnicastRemoteObject` means the class cannot inherit from another class
- Common solution is to use Veneers:
  - `Interface I extends Remote`
  - `Class C implements I extends something-else`
  - `Class V implements I extends UnicastRemoteObject`
  - only state in a V is a reference to a C
  - methods in V call corresponding methods in the referenced C object

Using the `rmiregistry` as a name server

- Name servers offer an advertise/lookup facility
  - advertise a name (string) and reference (remotely invocable object)
  - lookup a reference by providing the name
- `rmiregistry` works like this, but only accepts references to processes on the same machine
- care is needed over the `CLASSPATH` to ensure the `rmiregistry` can see the stubs etc
- the references handed out by the `rmiregistry` can be cast to the interface type, but not to the class type, because they actually point at a local stub object
- be aware that the client has to know the `rmiregistry` used by the server, and they have to agree on the name used in the advertise/lookup operations
- would usually build your own, more flexible, name server; just use `rmiregistry` to access that name server

Practical Activity

- Look in `/users/students4/software/public/GCM/`
  Look in the `PD-software/simple-rmi-example` subdirectory
- You will find an instruction sheet (in `.tex` `.dvi` `.ps` & `.pdf` formats)
- You will also find five Java files; these form a whole system, copy them
- Follow the instructions very thoroughly and carefully
- You should probably do this on Linux rather than Windows

- This is essential preparatory work for the RMI assessment issued shortly
  - You cannot afford to defer your learning, do this exercise asap (and certainly by the end of the weekend)
- If you've taken DAS4, you've already done this exercise;
  but refresh your memory and practice your RMI coding anyway