SCC.311: Remote Invocation

Prof. Barry Porter

Coursework…

- Our lab stream starts on Monday
- Lab work is based on Java, using RMI as an example middleware
- We've posted an "RMI primer" along with the first coursework stage
- This stage is due for submission on Friday of week 3

Overview

- Remote invocation is the process of executing a piece of logic on a different computer, using a particular protocol

- Message exchange is handled through an agreed protocol
  - The semantics of this protocol depend on the context of your application
  - Different protocols will offer different reliability, scalability, and performance

- The implementation of this protocol, and any associated tools, is a *communication middleware*

Protocol styles

Style	*Messages sent by*		
	Client	**Server**	**Client**
R	Request	-	-
RR	Request	Reply	-
RRA	Request	Reply	Acknowledgement

- R: no value needs to be returned from the server / no confirmation is needed; client can "fire and forget" in a non-blocking way
- RR: typical "request-reply" protocol: if reply from server is lost in transit, request may be repeated by client
- RRA: server needs to know the client got its reply, e.g. to allow resources to be released or coordinate with other communications

Basic remote invocation (client)

```
const int SENSOR_READING = 1

data Header {
   int msgType
   int payloadSize
}

data SensorInfo {
   int reading
}
```

```
TCPSocket socket = new TCPSocket()
socket.connect(“143.94.13.8”, 2945)
```

Connect to a remote computer

```
Header h = new Header()
h.msgType = SENSOR_READING
h.payloadSize = size(SensorInfo)
socket.send(h)
```

Send generic message header

```
SensorInfo info = new SensorInfo()
info.reading = mySensor.getReading()
socket.send(info)
```

Send operation-specific data

```
socket.disconnect()
```

Clean up

Basic remote invocation (server)

```
const int SENSOR_READING = 1

data Header {
    int msgType
    int payloadSize
}

data SensorInfo {
    int reading
}
```

```
TCPServer server = new TCPServer()
server.bind("localhost", 2945)
```

Prepare to receive client connections

```
while (true) {
  TCPSocket sock = server.accept()
  if (client != null) {
    byte buf[] = sock.recv(size(Header))
    Header hdr = (Header) buf
    buf = sock.recv(hdr.payloadSize)
    if (hdr.msgType == SENSOR_READING) {
      SensorInfo info = (SensorInfo) buf
      saveSensorData(info, sock.addr)
    }
    sock.disconnect()
  }
}
```

Receive generic message header

Receive operation-specific data

Do something

Client

Agree on TCP
Agree on network packet format
Agree on message types

Server

```
TCPSocket socket = new TCPSocket()
socket.connect("143.94.13.8", 2945)

Header h = new Header()
h.msgType = SENSOR_READING
h.payloadSize = size(SensorInfo)
socket.send(h)

SensorInfo info = new SensorInfo()
info.reading = mySensor.getReading()
socket.send(info)

socket.disconnect()
```

```
TCPServer server = new TCPServer()
server.bind("localhost", 2945)

while (true) {
  TCPSocket sock = server.accept()
  if (sock != null) {
    byte buf[] = sock.recv(size(Header))
    Header hdr = (Header) buf
    buf = sock.recv(hdr.payloadSize)
    if (hdr.msgType == SENSOR_READING) {
      SensorInfo info = (SensorInfo) buf
      saveSensorData(info, sock.addr)
    }
    sock.disconnect()
  }
}
```

Remotely invoked logic

Agree on TCP
Agree on network packet format
Agree on message types

Style	
R	✓
RR	
RRA	

Client

```
TCPSocket socket = new TCPSocket()
socket.connect(“143.94.13.8”, 2945)

Header h = new Header()
h.msgType = SENSOR_READING
h.payloadSize = size(SensorInfo)
socket.send(h)

SensorInfo info = new SensorInfo()
info.reading = mySensor.getReading()
socket.send(info)

socket.disconnect()
```

Server

```
TCPServer server = new TCPServer()
server.bind(“localhost”, 2945)

while (true) {
  TCPSocket sock = server.accept()
  if (sock != null) {
    byte buf[] = sock.recv(size(Header))
    Header hdr = (Header) buf
    buf = sock.recv(hdr.payloadSize)
    if (hdr.msgType == SENSOR_READING) {
      SensorInfo info = (SensorInfo) buf
      saveSensorData(info, sock.addr)
    }
    sock.disconnect()
  }
}
```

Remotely invoked logic

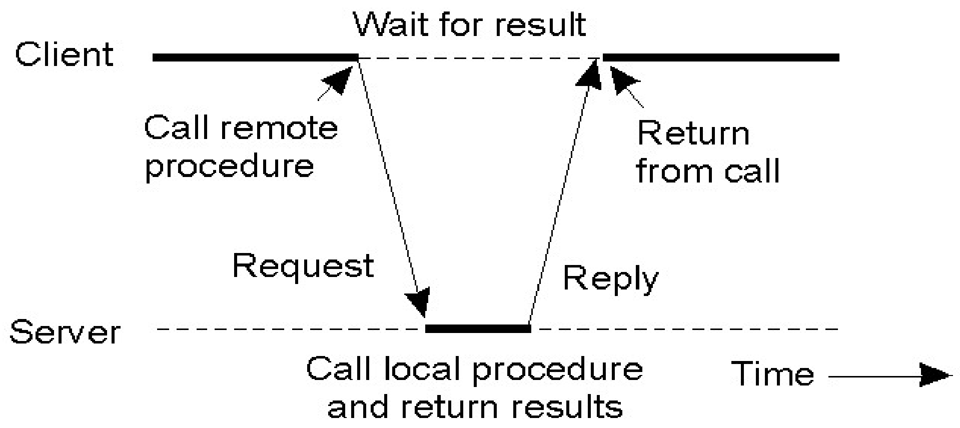
RPC

Remote invocation middleware

Writing everything at the level of message types can be tedious, and often obscures the application logic within lots of message- and network-handling code
 - *maybe we can do better than this?*

Java RMI	CORBA	WebServices
Protocol Buffers	HTTP / REST	(etc.)

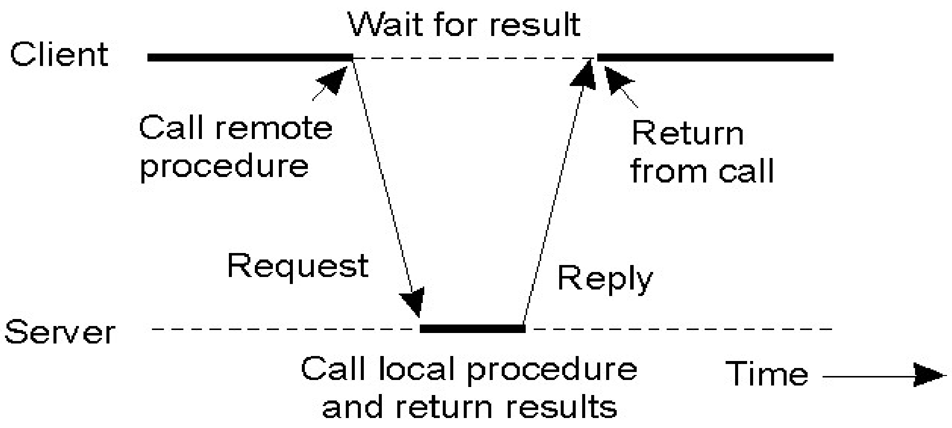
What is RPC?



- Remote Procedure Call

- In essence, the idea is:
  - it's nice that we can define and call functions for local programs
  - why not extend this to a distributed system, so that we can call an apparently local function and that function call actually happens on a remote computer?

What is RPC?

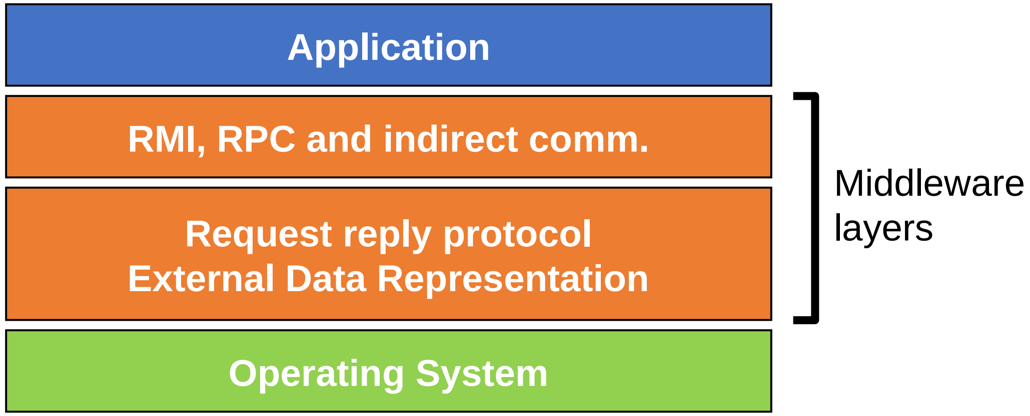


- Remote Procedure Call

- One of the simplest forms of communication middleware
  - Provides a high-level request-response mechanism to build distributed apps

- Usually *synchronous*, meaning that the client *blocks* while waiting for the procedure (cf. function) call to complete

RPC and Middleware



- Examples: XML-RPC, JSON-RPC, SOAP
- Interaction between processes is done using defined *interfaces*

Programming with Interfaces

- Separation of interface and implementation

```
interface Calculator:
    int add(int a, int b)
    int mul(int a, int b)
    int factorial(int a)
    bool isPrime(int a)
```

```
Class MyCalc implements Calculator {
    int add(int a, int b) {
        return a + b
    }
}
```

- In distributed systems we may use an IDL (Interface Definition Language): a language-independent **notation** of parameters / types
- Client software does not need to know the details of the implementation, cf. abstraction

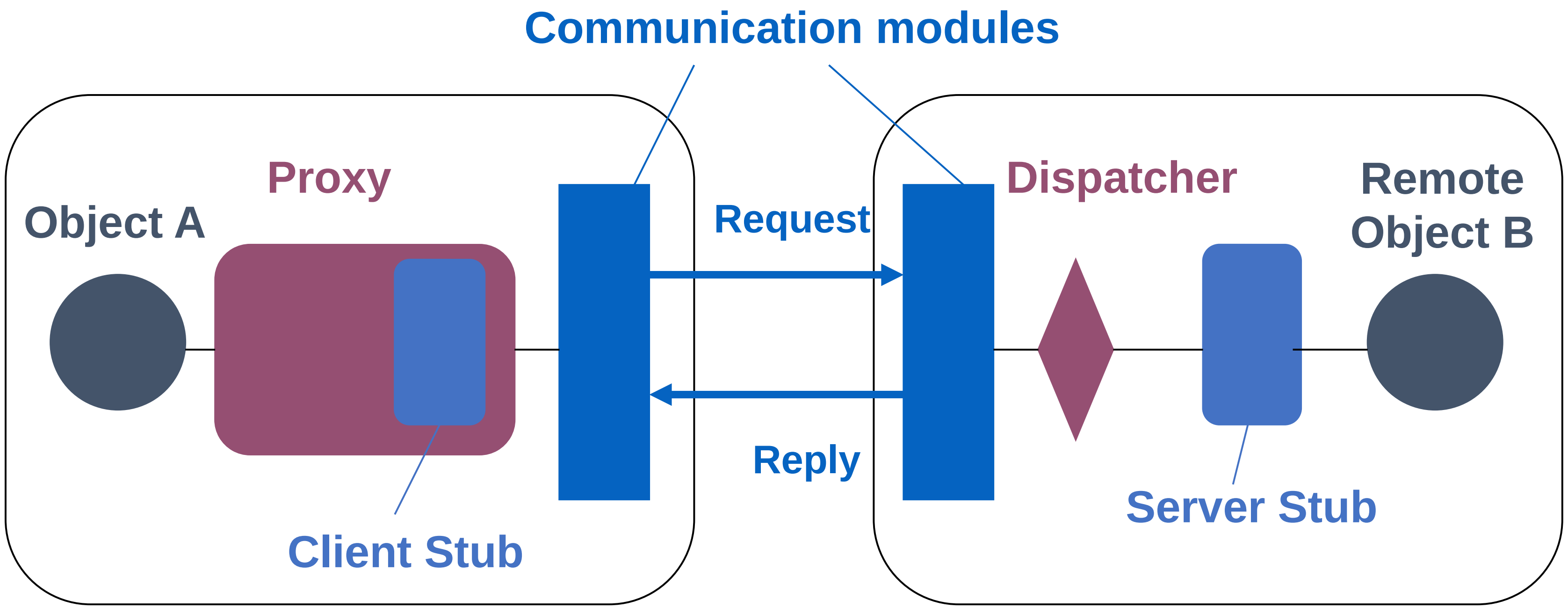
Programming with Interfaces

- Now that we have a separated interface, we can change our implementation to be a *proxy* for remotely-located code

```
interface Calculator:
    int add(int a, int b)
    int mul(int a, int b)
    int factorial(int a)
    bool isPrime(int a)
```

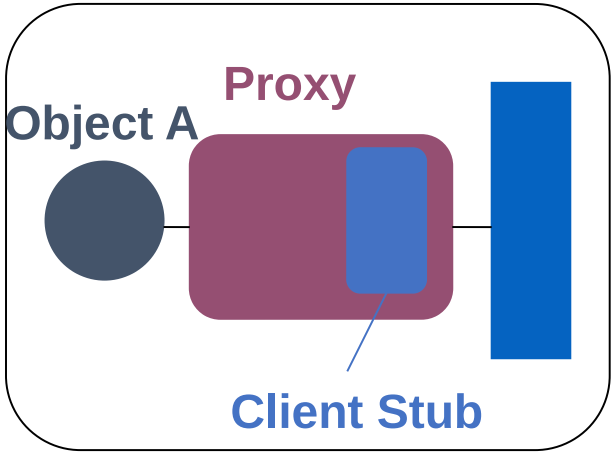
```
Class MyCalc implements Calculator {
    int add(int a, int b) {
        TCPSocket s = new TCPSocket()
        s.connect("200.34.56.98", 8739)
        s.send(new Header(ADD))
        s.send(new AddData(a, b))
        Result r = s.recv()
        return r.value
    }
}
```

Implementing RPC



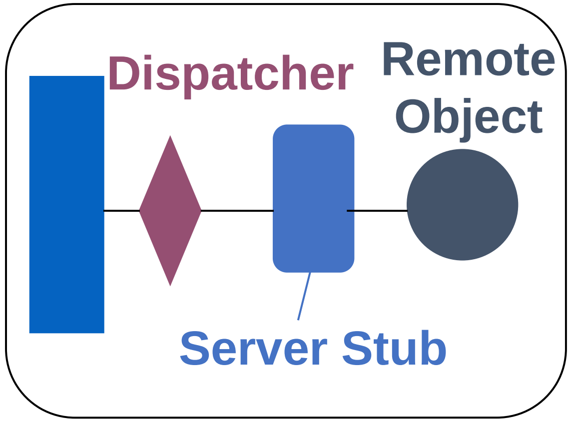
N.B. Proxies, stubs, dispatchers are generated automatically by an appropriate IDL compiler

Key components: client side



- Proxies
  - Masquerade as a local version of the remote interface
  - Redirect calls to client stubs
  - May perform other actions (see smart proxies)
- Client stub
  - Carries out marshalling (flattening) of a call into a request message sent to remote end
  - Also unmarshalls returning replies
  - One stub per interface procedure

Key components: server side



- Dispatchers
  - Receive incoming messages and direct them to an appropriate server stub

- Server stubs (skeletons)
  - Unmarshalls message and then invokes appropriate code body
  - Also marshals reply values and initiates transmission back to the client

What we get from this...

```
thing = RPCService.getRemote(serverName);

thing.callFunction("hi")
```

After the initial acquisition of this entity, from the middleware, use of it then looks exactly like a normal local function call

…but there might be a dragon or two…

- A lot of our current understanding on the theory of RPC comes from the original researchers and designers of the Java language[1]
  - Remote calls have different latency to local calls
  - Memory access models are different if we pass references around
  - Partial failures are possible

[1] *J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical report, 1994*

…middleware and the network

- What is really happening when we do RPCs?

```
thing.callFunction("hi")
```

client

1: Open a new TCP connection
2: send my RPC message, with parameters etc.
3: wait for the server to send me the result
4: close the TCP connection

server

```
thing.callFunction("hi")
```

client

1: Open a new TCP connection
2: send my RPC message, with parameters etc.
3: wait for the server to send me the result
4: close the TCP connection

server

Protocol guarantees

- What delivery guarantees does the exchange protocol give?
  - Referred to as ‘call semantics’ in the book

- Local procedure calls = ‘exactly once’ guarantee

- But for RPC?
  - Different guarantee types are possible depending on the protocol implementation

Focus on the underlying protocol

- Let's focus on the communications module for RPC which will provide a protocol that mimics the semantics of a local call
  - For the sake of this discussion let's assume the underlying protocol is **UDP**
  - (note RPC is more commonly implemented with TCP in modern middleware)
- Problems
  - Request message may get lost
  - Reply message may get lost
  - Client may crash
  - Server may crash

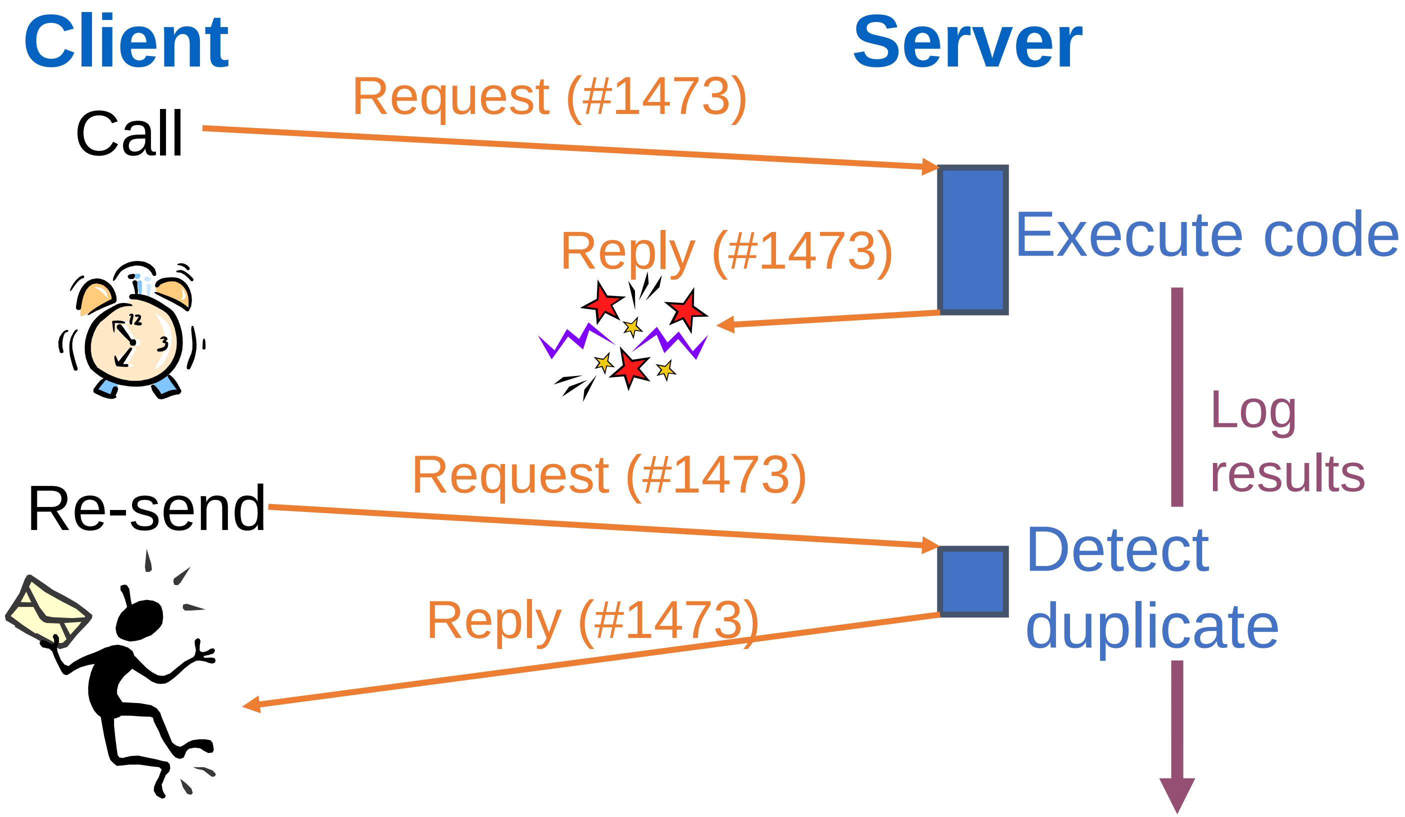
Lightweight protocol semantics

- Maybe semantics
  - Send request to server; which sends back a reply
  - No guarantees at all if anything goes wrong

- At least once semantics
  - Sends message and if reply not received after a given time, the message is re-sent (failure assumed after *n* re-sends)
  - Will guarantee the call is made “at least once”, but possibly multiple times
  - Ideal for idempotent operations (i.e. same effect)

Lightweight protocol semantics

- At *most* once semantics



Lightweight protocol semantics

- Local procedure calls have an even stronger exactly once semantic
- So far, for RPC, we have:

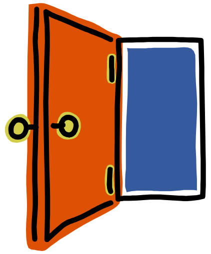
Semantics	*Fault tolerance measures*		
	Retransmit request	**Duplicate filtering**	**Re-execute procedure or retransmit reply**
Maybe	No	No	N/A
At least once	Yes	No	Re-execute procedure
At most once	Yes	Yes	Retransmit reply

RPC protocol semantics

- Exactly once semantics
  - In this case the procedure will be carried out once (completely) or not at all (operation aborted)

- This builds on the "at most once" protocol, but also adds support for atomicity

- We'll cover this topic in our lectures on fault tolerance and dependability



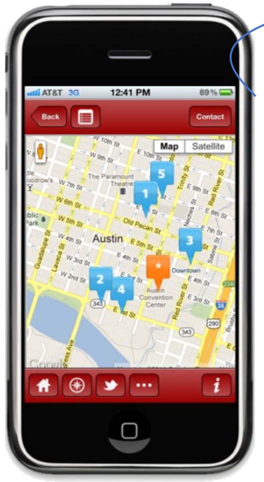
From RPC to RMI

- Remote Method Invocation (RMI) is the Java-specific built-in middleware technology which implements the RPC concept, integrating it seamlessly with the Java language

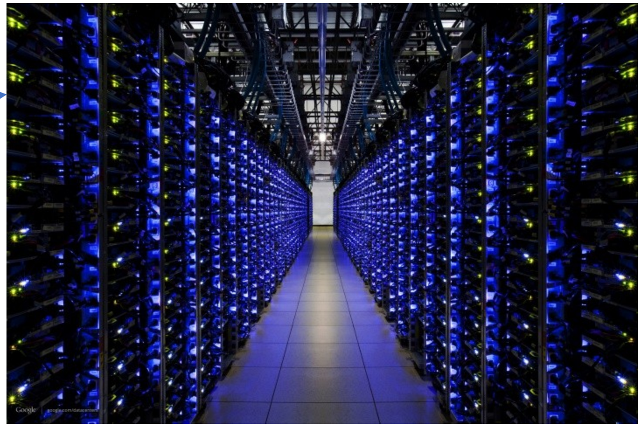
- RMI implements remote objects in an almost transparent way: you can pass object references into remote function calls to create complex object reference graphs which span continents
  - The "almost" part is that RMI chooses to expose a new class of exceptions on all remote calls, via *RemoteException*, which must be caught in the caller

RMI Basics

Example: You need to develop a Java mobile app to access the Google Maps Service



```
List<Obj> nShops =
GoogleMaps.nearestShops(x, y);
```

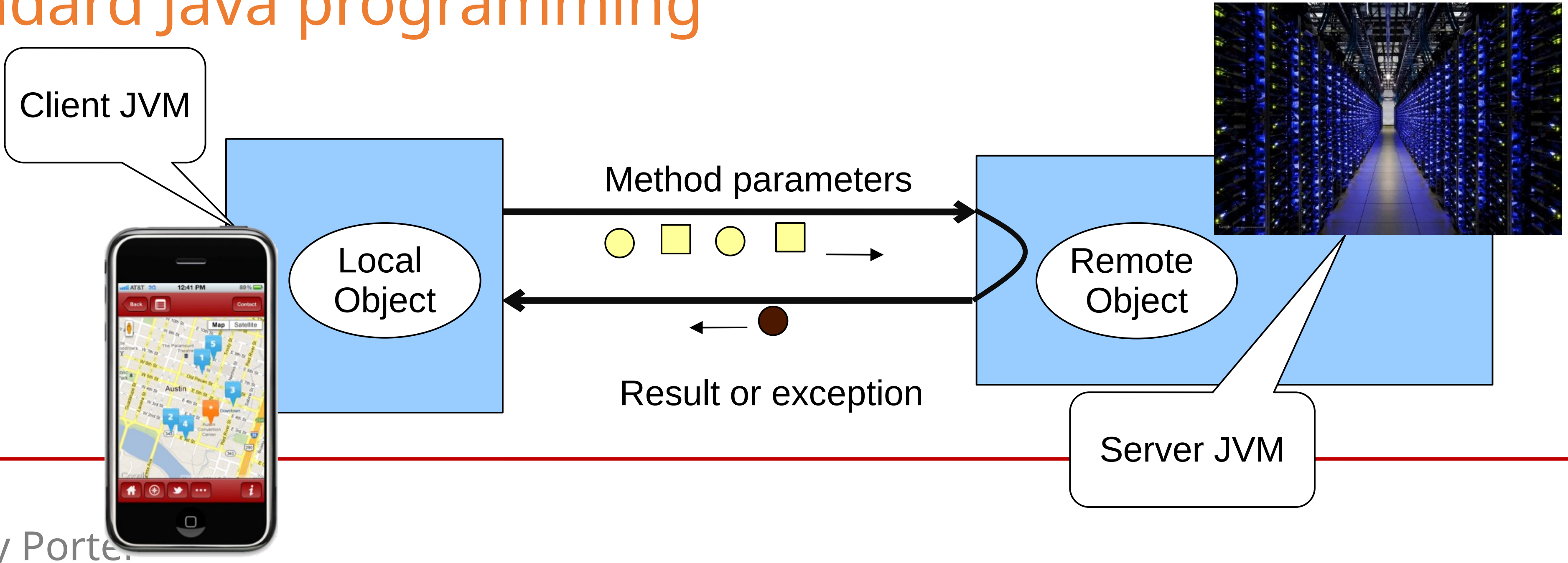


Google Servers @ Dublin

Your Mobile @ Lancaster

RMI Basics

- RMI allows one Java object to call methods on another Java object in a different JVM
- The intention is to make distributed programming as easy as standard Java programming



RMI Basics

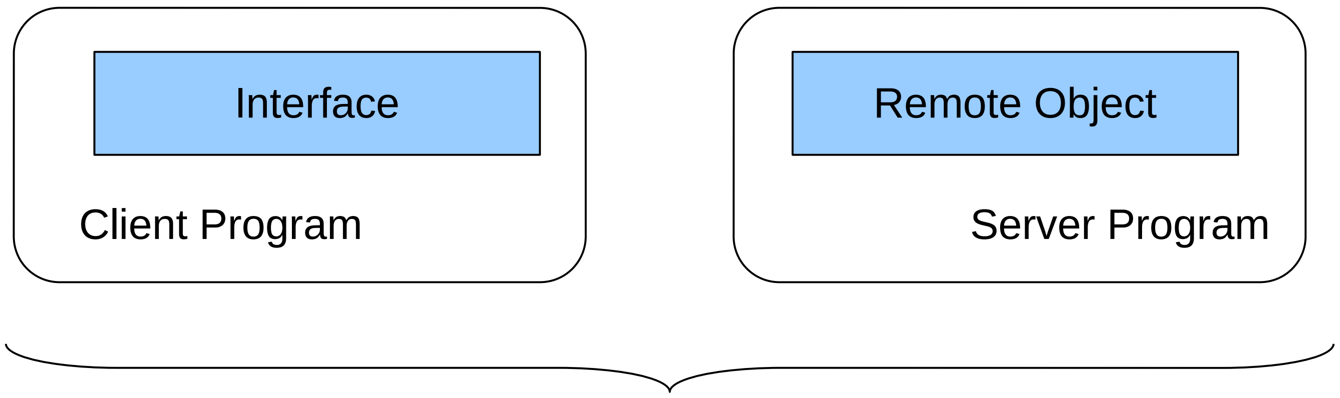
- RMI uses *interfaces* to specify a remote object: we define an interface which extends from **java.rmi.Remote**
- We define a class which implements this interface; we can then instantiate an object from this class which can be advertised for remote access
- A client program only needs access to the interface type (*not the implementing class*), and can then acquire a reference to the remote object of this type via the RMI middleware service

RMI Basics

- The advertisement (at the server) and lookup (at the client) of remote objects is done through a special service called the *RMI registry*
  - We execute the registry at the command line using the command *rmiregistry*

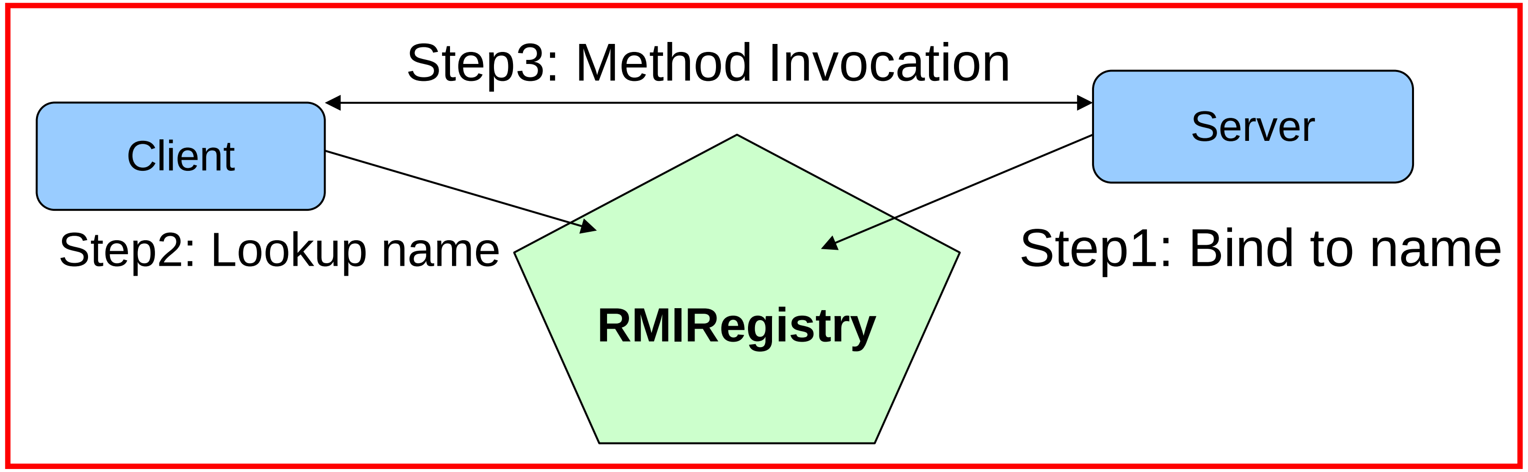
- This service associates *names* with *object references*
  - The registry often runs on the same host as a server system, but does not need to
  - A server and client both need to talk to the same registry service, on the same host, to advertise and look up a named object

RMI Basics



naming.lookup("rmi://www.google.com/MapService")

naming.rebind("rmi://www.google.com/MapService", RemoteObjectReference)



HTTP/REST

REST

- ...and now for something different!
- So far we've covered remote procedure call and RMI
- These are not the only forms of remote invocation

REST

- Representational State Transfer (REST) is a set of resource-oriented architectural principles
  - RPC/RMI are operation-/transaction-oriented
  - RPC: `readStudent(1234)`
  - REST: `GET /students/1234`
- Properties:
  - Every resource is addressable using a Uniform Resource Identifier (URI)
  - HTTP-based: basic HTTP verbs and status codes (universal interface)
  - Self-descriptive: responses include description and next step(s) links
  - **Stateless:** data required to transition between states is in each request (via cookies)

HTTP Methods (verbs)

- The universal/uniform interface of REST

Verb	CRUD	Idempotent?
POST	Create	✘
GET	Read	✓
PUT	Update/Replace	✓
PATCH	Modify	✘
DELETE	Delete	✓

HTTP Methods (verbs)

- Read specific student
  `GET /students/1234`
- Read all students
  `GET /students`
- Create a student
  `POST /students`
- Update specific student
  `PUT /students/1234`
- Delete specific student
  `DELETE /students/1234`

HTTP Methods (verbs)

- Read specific student email address(es)
  `GET /students/1234/email`
- Update specific student email address
  `PUT /students/1234/email/1`
- Delete specific student email address
  `DELETE /students/1234/email/2`

HTTP Content Types

- The content type used by each verb, in both the request and the response message, is configurable
- This is done using headers which can be included in the request and response
  - Common content types are `text/html`, `text/xml`, `text/json`, `image/jpeg`, etc.
  - The sender of a request can also specify the content types that it is expecting and can process, as part of its request message

HTTP Status Codes

1xx Informational

2xx Success

200 Resource was read, updated, or deleted

201 Resource was created

3xx Redirection

301 Resource has permanently moved to a new URI

4xx Client Error

400 Bad request

403 Not authorized to perform this action

404 Resource not found

5xx Server Error

HTTP Semantics

- The most obvious feature of REST is that servers do not hold any per-client state

- Instead, the client sends the current state with every request (this is what cookies are)

- This allows servers to consume fewer memory resources, and also allows a client request to hit any server, because the request carries all of the state

HTTP Semantics

- REST, when implemented as originally intended, also has a general assumption of *idempotence*, meaning that an operation will only ever have a single effect (repeating the same operation, with the same state, has no effect)
  - Keeps servers slender
  - Very useful in distributed environments
    - Multiple ‘servers’
    - Unreliable network

HTTP Summary

- Because it is a text-based format, is very simple, and is not language-specific, HTTP has become a kind of general interoperability protocol
- A wide range of other protocols have been designed which can operate on top of HTTP, taking advantage of this common carrier
- Linking back to RPC, the **Web Services** framework is a language-independent RPC solution which is built on top of HTTP

Further reading

- **"Distributed Systems: Concepts and Design" (CDKB), ch 5**
  - also optionally ch 4 for background
- TvS, pp. 145-158, 68-98, 99-134
- REST API Tutorial: https://www.restapitutorial.com/