# RMI

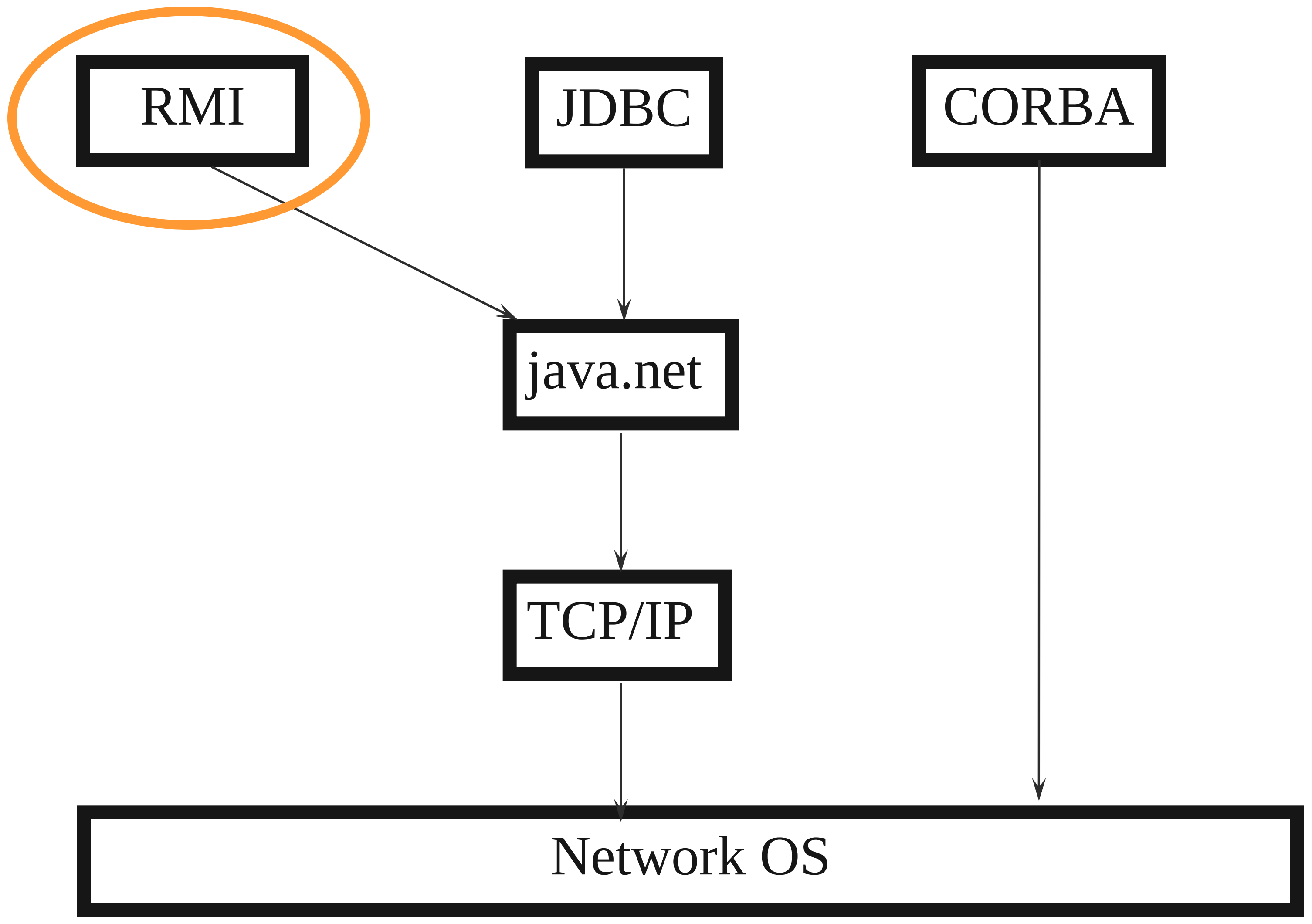
HSBC Technology and Services

HSBC
The world's local bank

## Introduction

- Java
- Networking
- Distributed Computing

## Overview



## What Is RMI?

- Access to Remote Objects
- Java-to-Java <u>only</u>
- Client-Server Protocol
- High-level API
- Transparent
- Lightweight

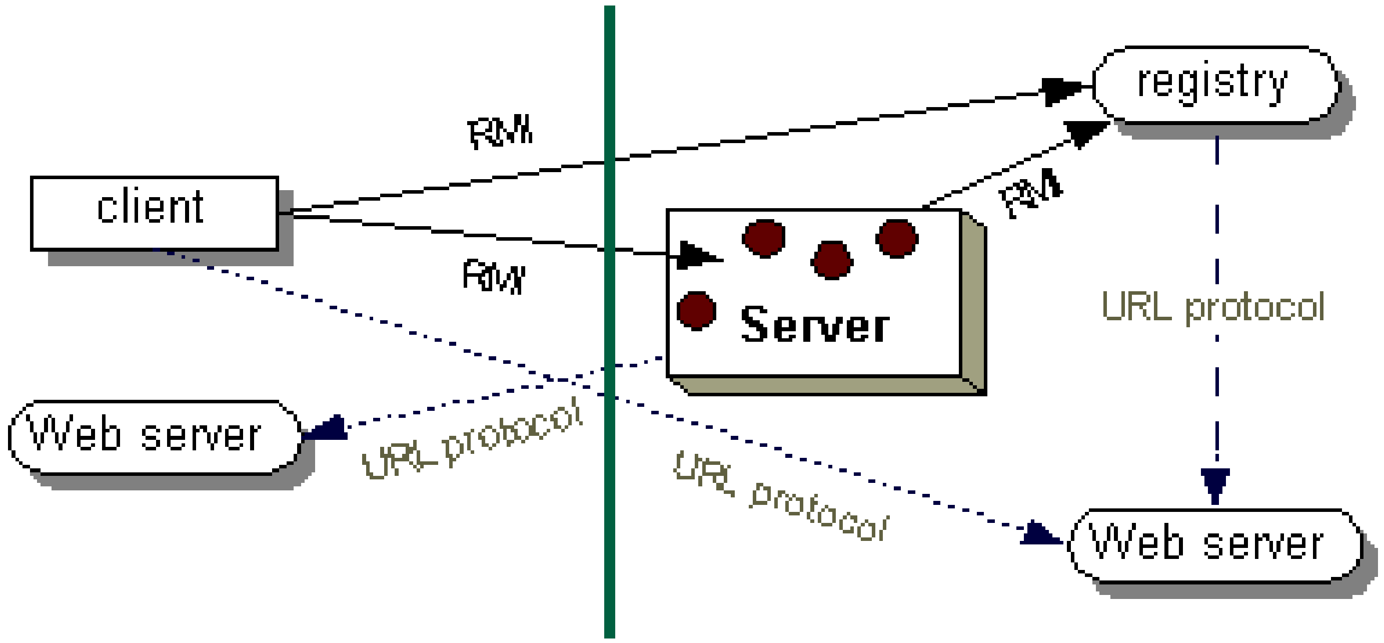
## Examples of Use

- Database access
- Computations
- Any custom protocol
- <u>Not</u> for standard protocols (HTTP, FTP, etc.)

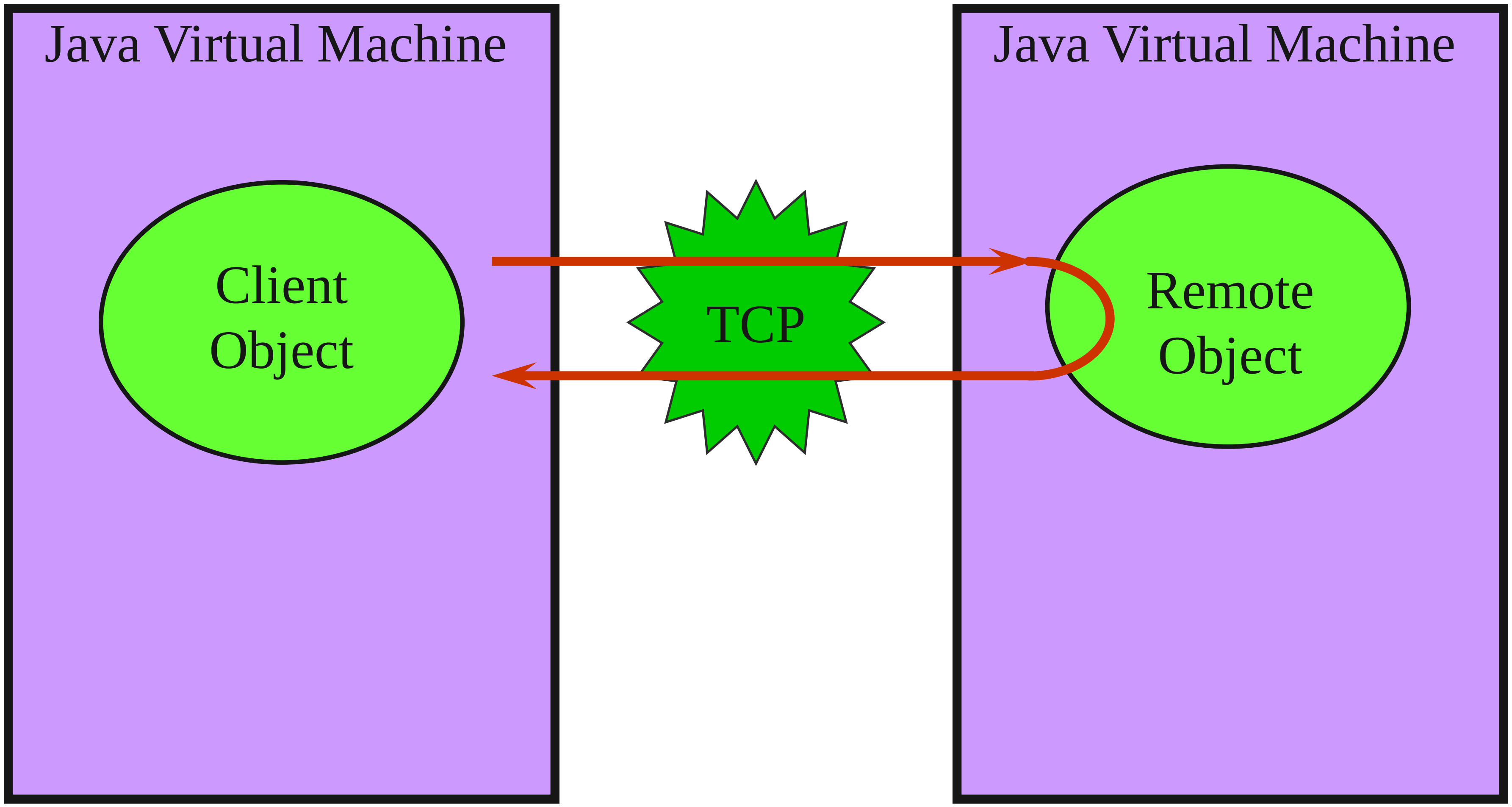
## Related Technologies

- **RPC** (“Remote Procedure Calls”)
  - Developed by Sun
  - Platform-specific
- **CORBA** (“Common Object Request Broker Architecture”)
  - Developed by OMG
  - Access to non-Java objects (as well as Java)
- **DCOM** (“Distributed Common Object Model”)
  - Developed by Microsoft
  - Access to Win32 objects
- **LDAP** (“Lightweight Directory Access Protocol”)
  - Finding resources on a network

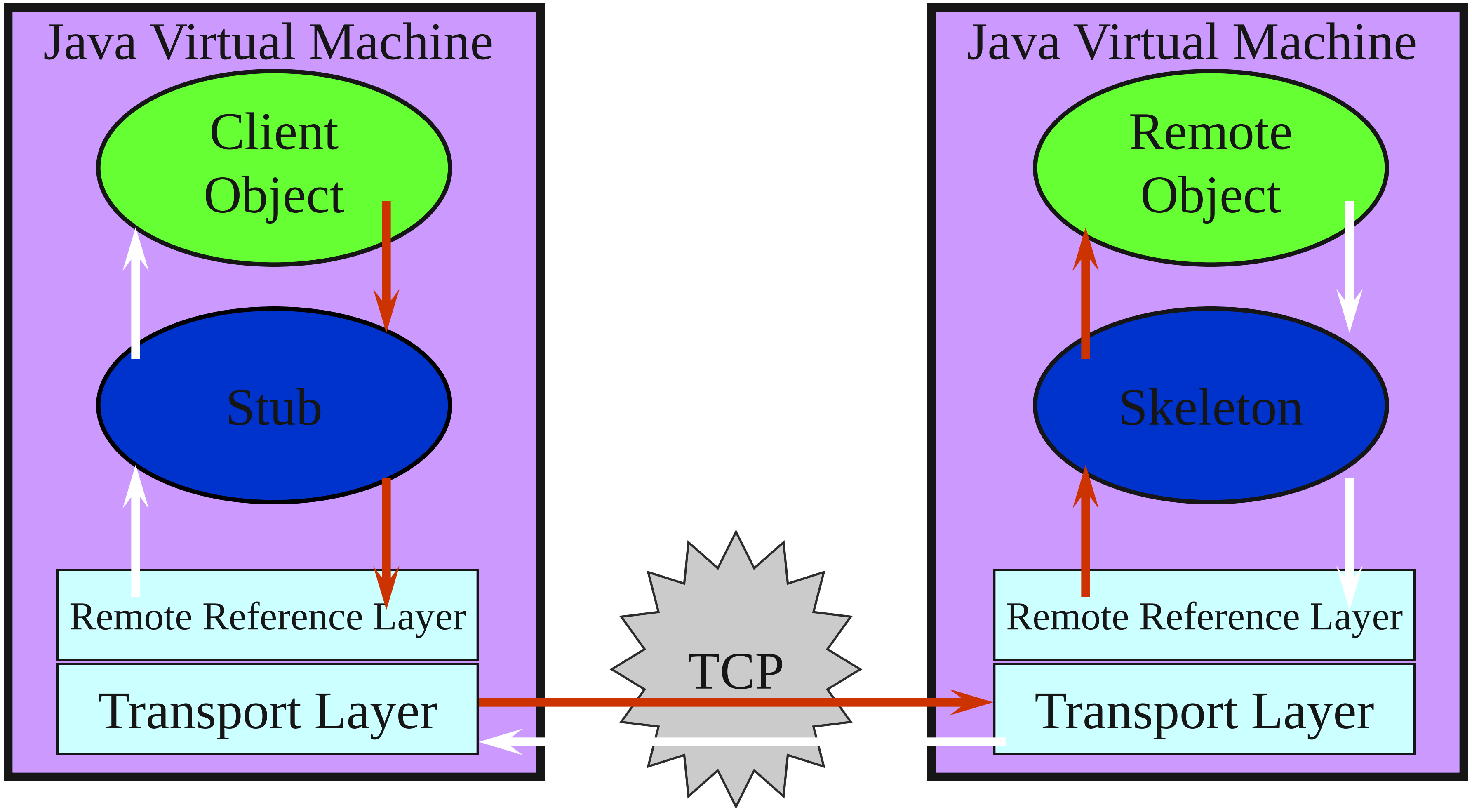
# Part I: RMI Concepts



## Remote Objects (Diagram)



## RMI Layers


Java Virtual Machine
Client Object
Stub
Remote Reference Layer
Transport Layer
TCP
Java Virtual Machine
Remote Object
Skeleton
Remote Reference Layer
Transport Layer

## Remote Objects

- Remote Objects
  - Live on server
  - Accessed as if they were local

## } Registries

- } Name and look up remote objects
- } Servers can register their objects
- } Clients can find server objects and obtain a remote reference
- } A registry is a <u>running process</u> on a host machine

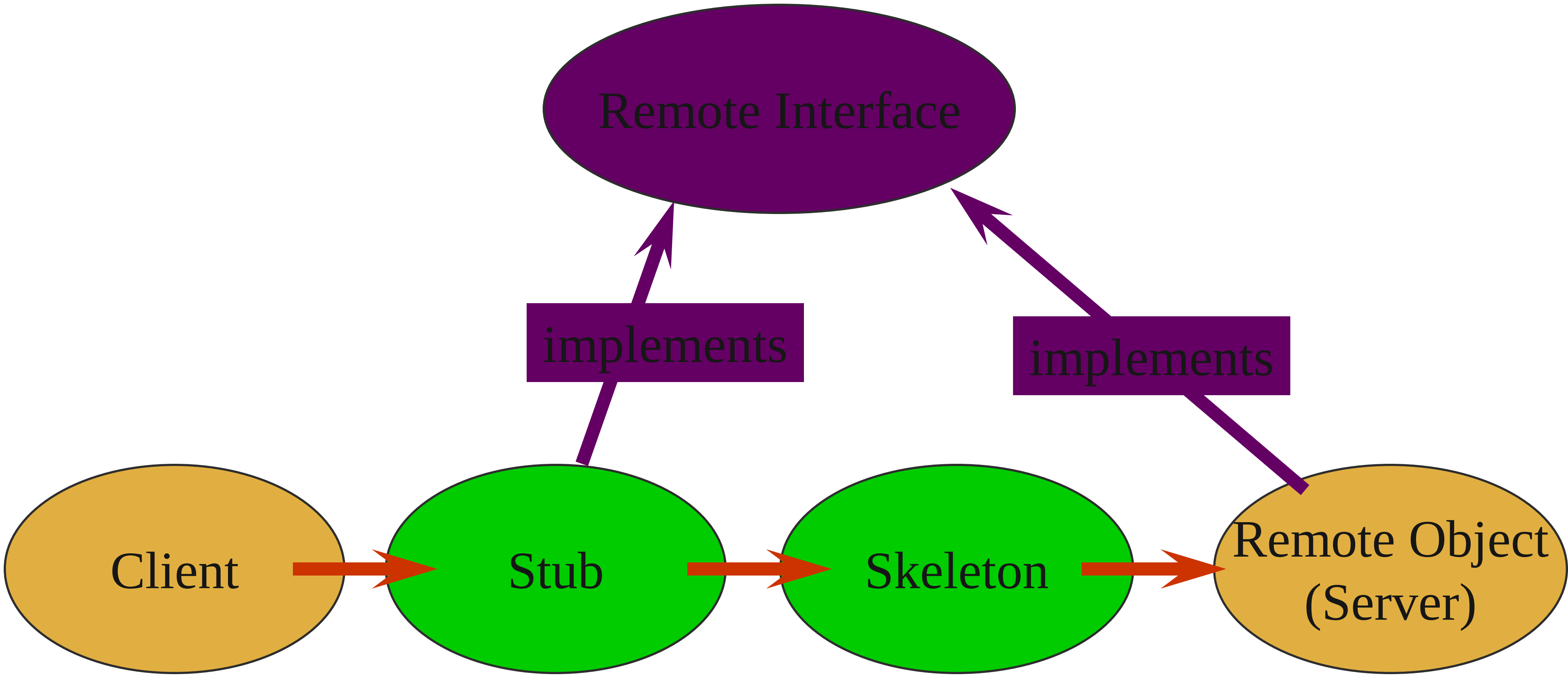
## Remote References and Interfaces

- Remote References
  - Refer to remote objects
  - Invoked on client <u>exactly</u> like local object references
- Remote Interfaces
  - Declare exposed methods
  - Implemented on client
  - Like a proxy for the remote object

## Stubs and Skeletons

- Stub
  - lives on client
  - pretends to be remote object
- Skeleton
  - lives on server
  - receives requests from stub
  - talks to true remote object
  - delivers response to stub

## Remote Interfaces and Stubs



## Remote Reference Layer

- Local pointer’s not good enough
- Figures out which remote object is being referenced
- Could span multiple virtual machines
- Communicates via TCP/IP

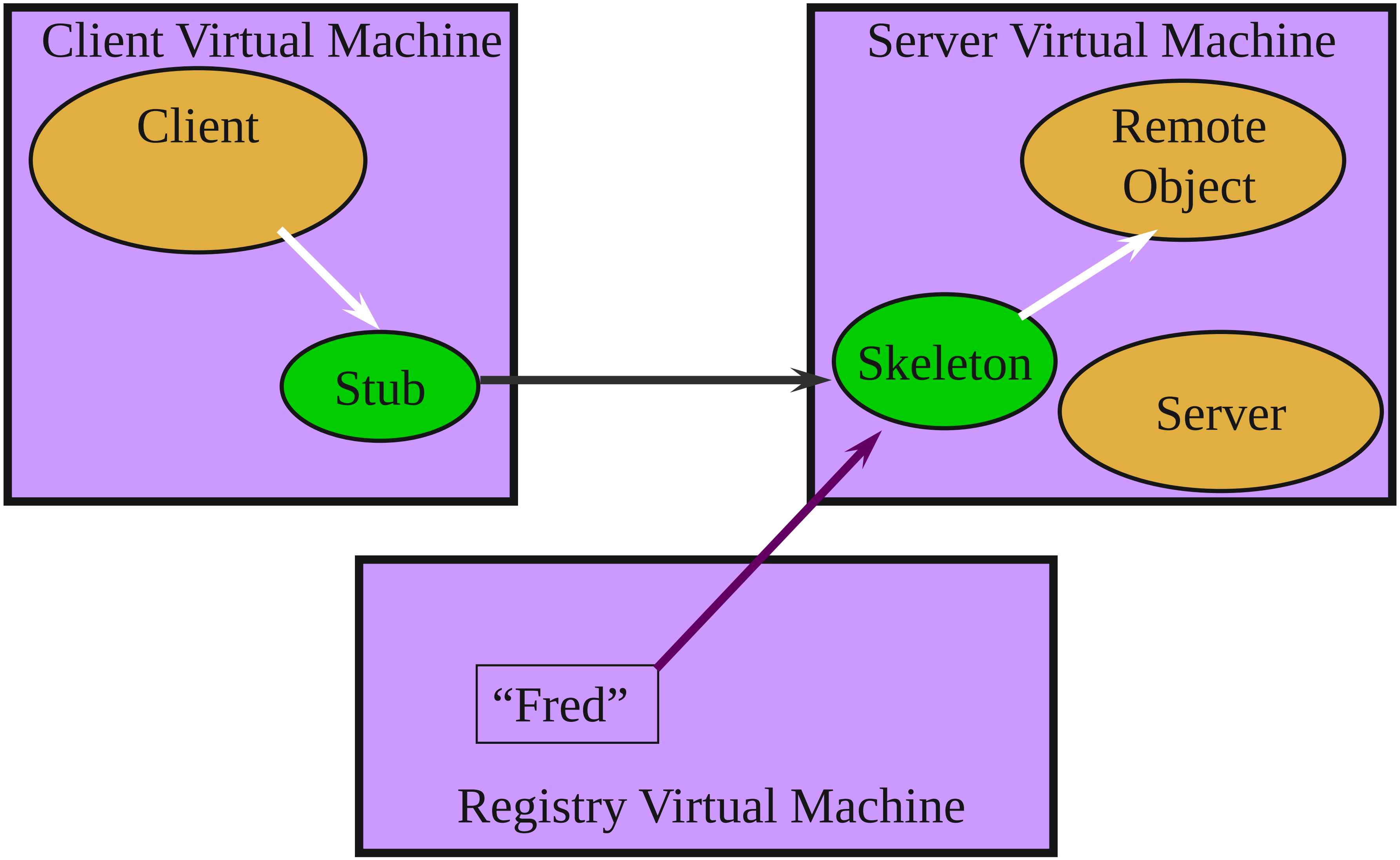
## Transport Layer

- Deals with communications
- Connection management
- Dispatching messages between stub and skeleton
- Distributed Garbage Collection
- Sits on top of java.net

## HTTP Tunneling

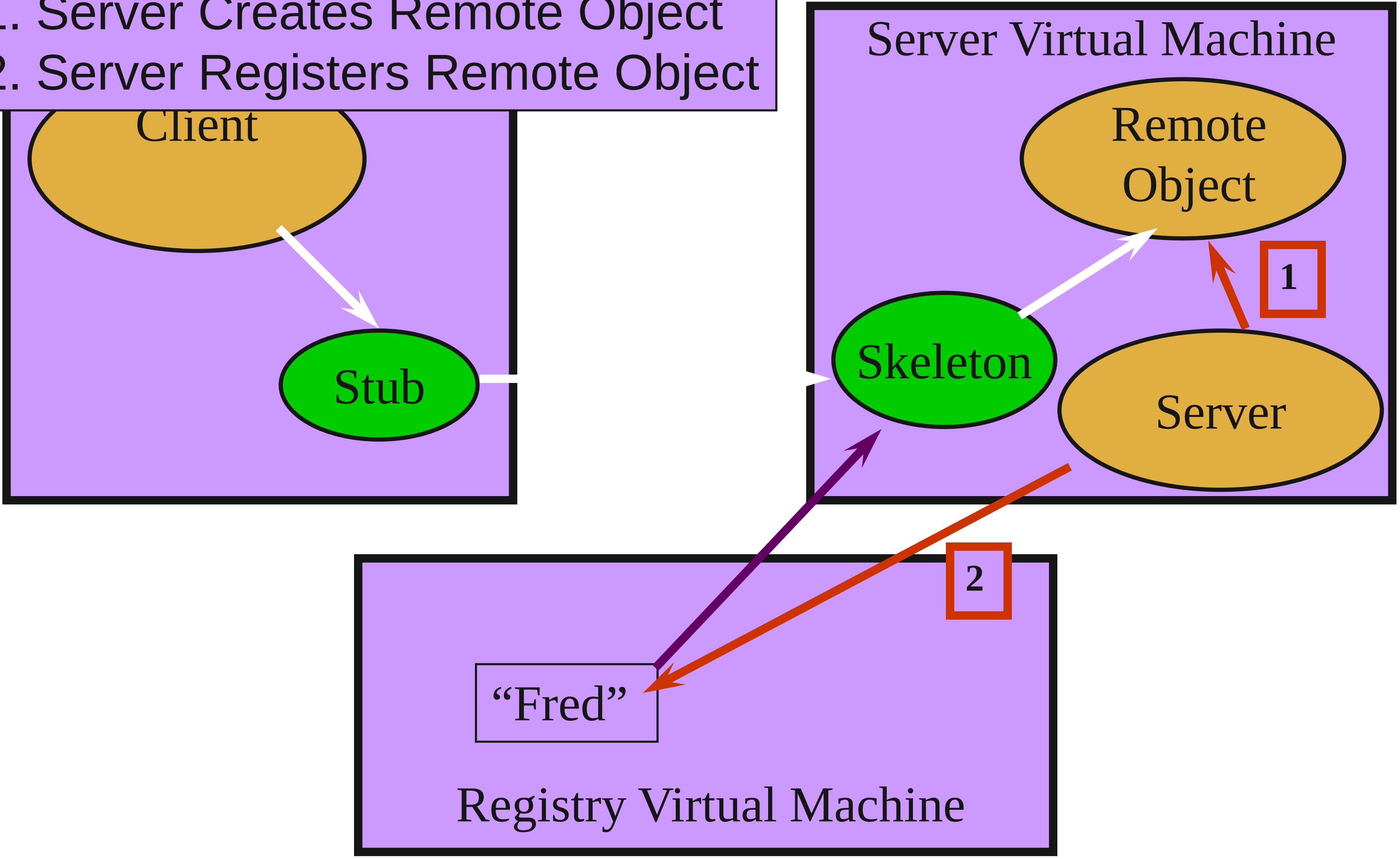
- Cool: if it can’t make the connection normally, it will tunnel through port 80
- Allows clients behind firewall to make remote calls to server
- Note: does not work server -> client

## RMI System Architecture

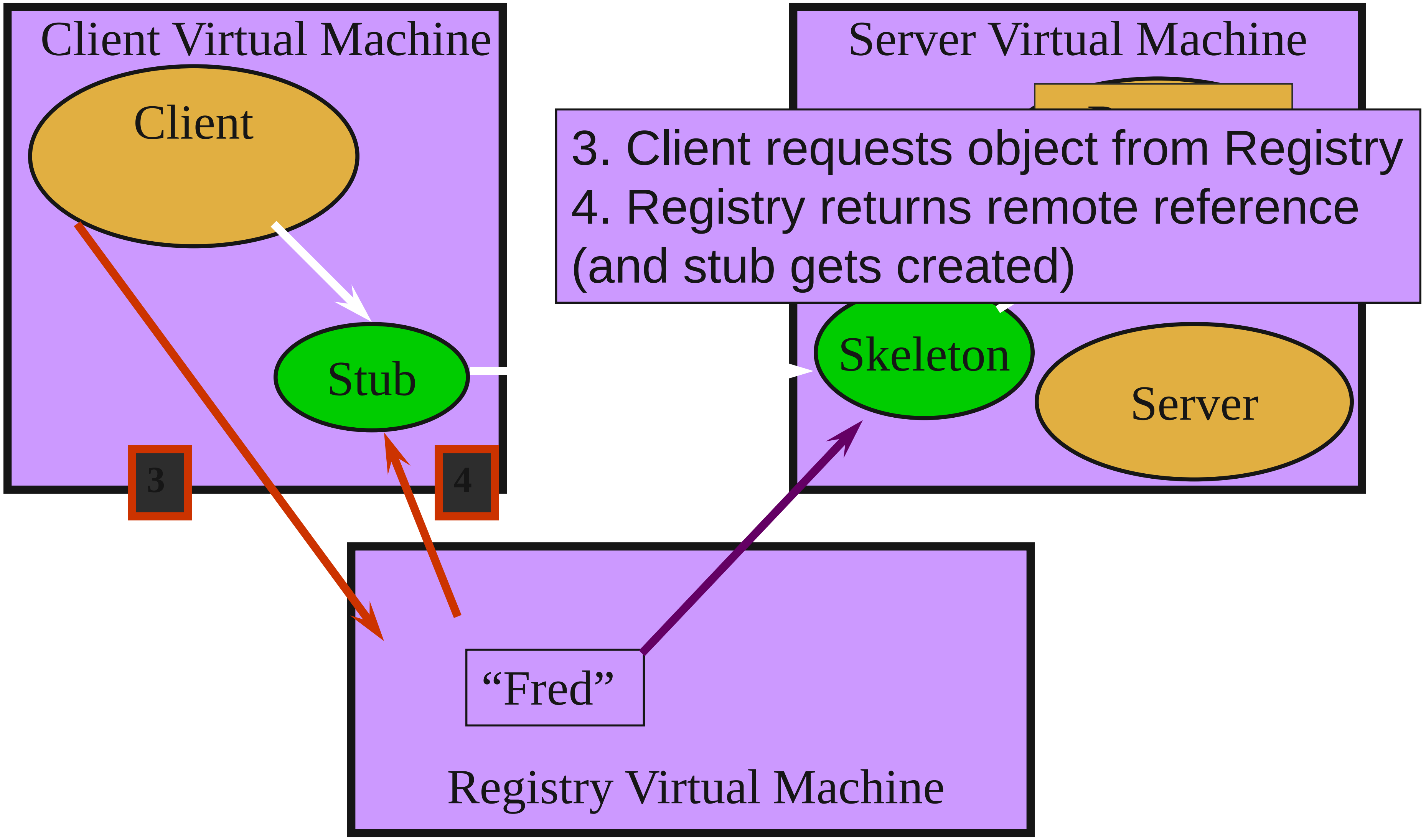


## RMI Flow

1. Server Creates Remote Object
2. Server Registers Remote Object

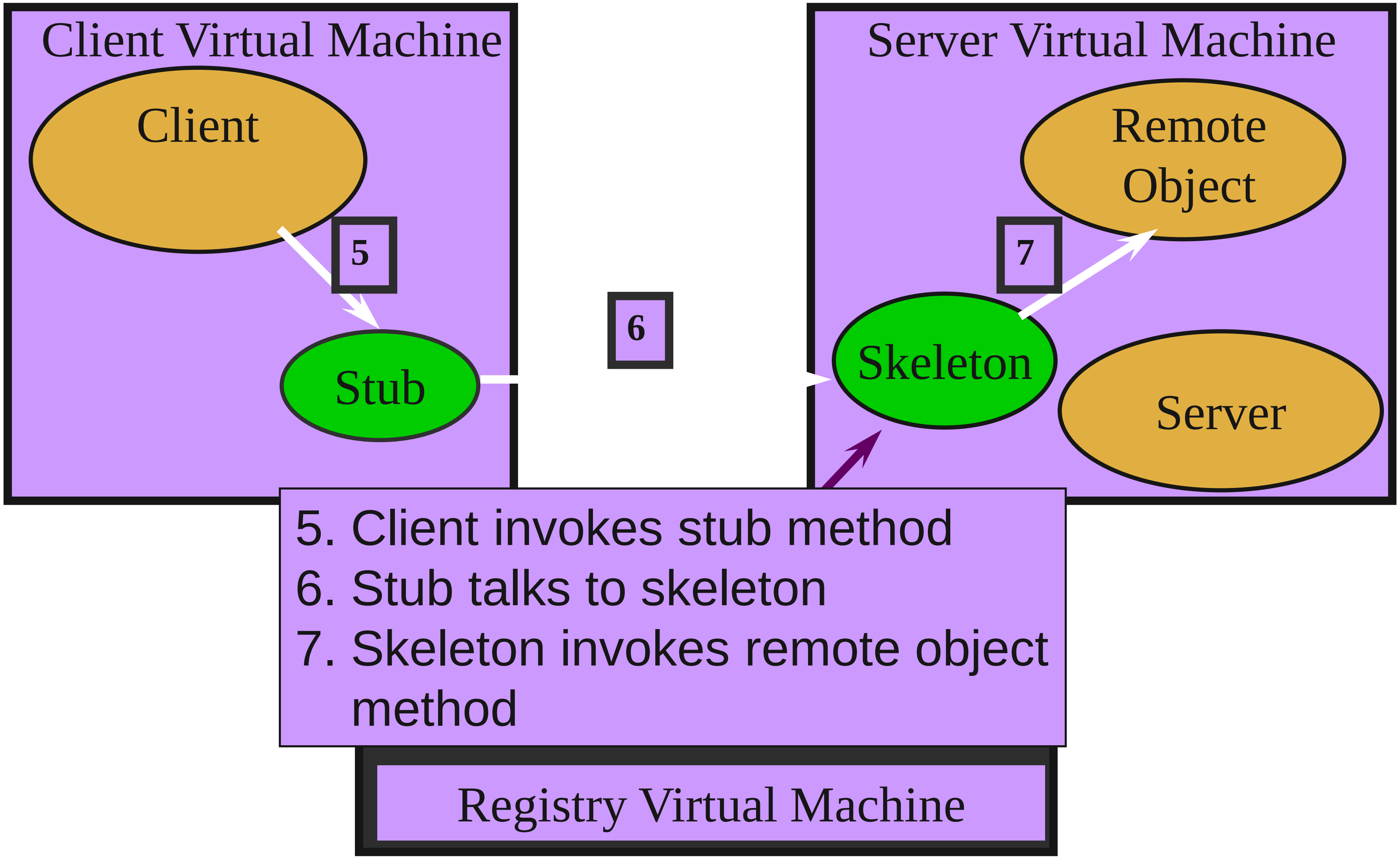


## RMI Flow



3. Client requests object from Registry
4. Registry returns remote reference (and stub gets created)

## RMI Flow



5. Client invokes stub method
6. Stub talks to skeleton
7. Skeleton invokes remote object method

## Part II: RMI Usage

## Creating Remote Objects

- Define a Remote Interface
  - extends java.rmi.Remote
- Define a class that implements the Remote Interface
  - extends java.rmi.RemoteObject
  - or java.rmi.UnicastRemoteObject

## Remote Interface Example

```
import java.rmi.*;
public interface Adder
  extends Remote
{
    public int add(int x, int y)
                throws RemoteException;
}
```

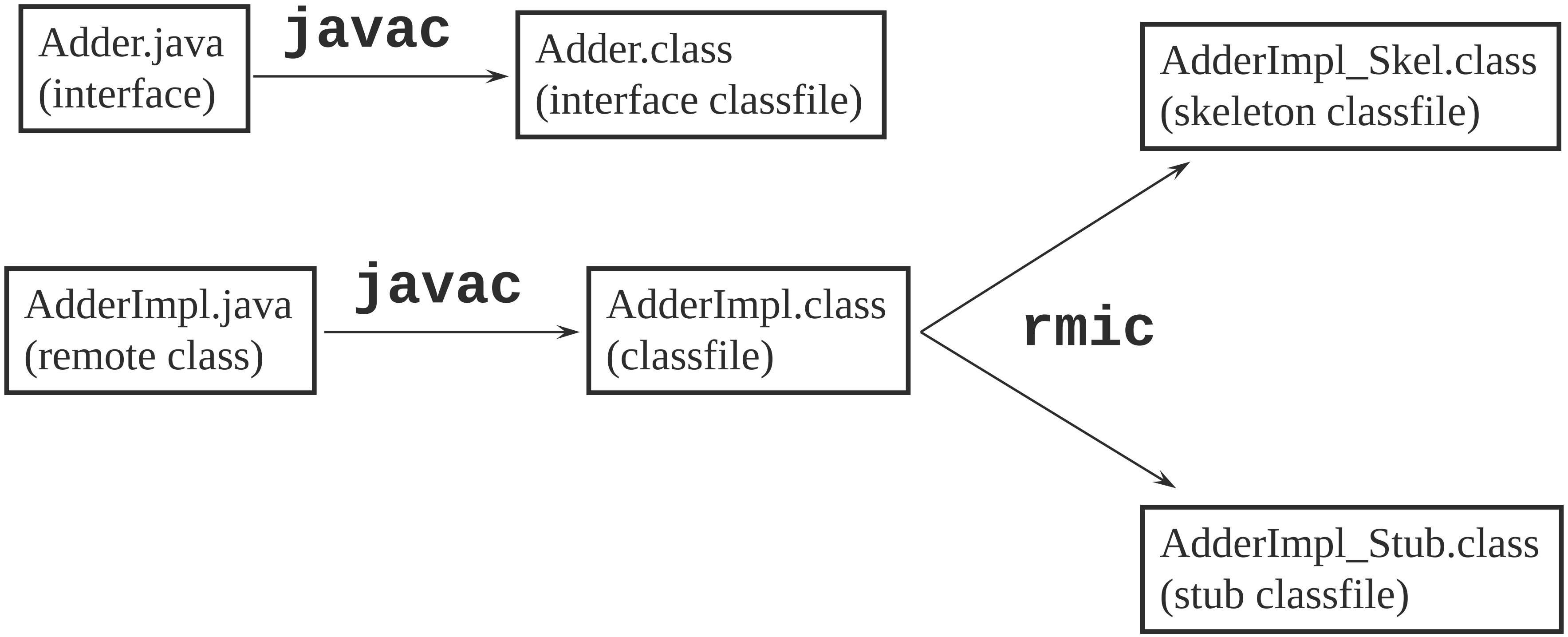
## Remote Class Example

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderImpl extends UnicastRemoteObject implements Adder
{
    public AdderImpl() throws RemoteException
    {
    }
    public int add(int x, int y)
        throws RemoteException
    {
        return x + y;
    }
}
```

## Compiling Remote Classes

- Compile the Java class
  - javac
    - reads .java file
    - produces .class file
- Compile the Stub and Skeleton
  - rmic
    - reads .class file
    - produces _Skel.class and _Stub.class

## Compiling Remote Classes (Diagram)



## Registering Remote Classes

- start the registry
  - running process
- Unix:

```
rmiregistry &
```

- Windows:

```
start /m rmiregistry
```

## Registry CLASSPATH

- Registry VM needs to be able to find stub file(s)
- You must set the CLASSPATH to include the directory containing the stub file
  - An easy way to check CLASSPATH is to use the javap command, supplying a fully package qualified class name. It uses the current CLASSPATH to find and print the interface to a class.
- Or, your server needs to specify the java.rmi.server.codebase System property (more later)

## Create the server

- Creates a new instance of the remote object
- Registers it in the registry with a unique name
- That’s it

## RMI Server Example

```
try {
    AdderImpl adder = new AdderImpl();
    Naming.rebind("adder", adder);
    System.out.println("Adder bound");
}
catch (RemoteException re) {
    re.printStackTrace();
}
catch (MalformedURLException me) {
    me.printStackTrace();
}
```

## Launch the Server

```
% java AdderServer &
Adder bound
```

## Server Logging

- invoke from command line

```
java
 -Djava.rmi.server.logCalls=true YourServerImpl
```

- or enable inside program

```
RemoteServer.setLog(System.err);
```

## Creating an RMI Client

- Install a Security Manager
  - to protect from malicious stubs
- Find a registry
  - use java.rmi.Naming
- Lookup the name, returns a reference
- Cast the reference to the appropriate Remote Interface
- Just use it!

## RMI URLs

`rmi://`*`host`*`[:`*`port`*`]/`*`name`*

- default port is 1099
- Specifies hostname of *registry*
- can also use relative URLs
  - name only
  - assumes registry is on local host

## RMI Client Example

```
System.setSecurityManager(
  new RMISecurityManager());
Adder a = (Adder) Naming.lookup("adder");

int sum = a.add(2,2);
System.out.println("2+2=" + sum);
```

## Remote Interfaces vs. Remote Classes

- Remember that the reference is to an interface
- You must make references, arrays, etc. out of the interface type, not the implementation type
- You can't cast the remote reference to a normal reference
- So name your Remote Objects with "Impl" (so you don't get confused)

## Parameter Passing

- Primitive types
  - passed by value
- Remote objects
  - passed by reference
- Non-remote objects
  - passed by value
  - uses Java Object Serialization

## Object Serialization

- aka Persistence
- saves the state (data) of a particular instance of an object
- *serialize* - to save
- *unserialize* - to load

## Java Serialization

- writes object as a sequence of bytes
- writes it to a Stream
- recreates it on the other end
- creates a brand new object with the old data

## java.io.Serializable

- Objects that implement the java.io.Serializable interface are marked as serializable
- Also subclasses
- Magically, all non-static and non-transient data members will be serialized
- Actually, it’s not magic, it’s *Reflection* (it’s done with mirrors)
- empty interface - just a marker
- It’s a promise

## Not All Objects Are Serializable

- Any object that doesn’t implement Serializable
- Any object that would pose a security risk
  - e.g. FileInputStream
- Any object whose value depends on VM-specific information
  - e.g. Thread
- Any object that <u>contains</u> a (non-static, non-transient) unserializable object (recursively)

## NotSerializableException

- thrown if you try to serialize or unserialize an unserializable object
- maybe you subclassed a Serializable object and added some unserializable members

## Incompatible Changes

- If class has members added or removed, it becomes <u>incompatible</u>
- java.io.InvalidClassException thrown if you try to deserialize an incompatible object stream

## Serial Version

- If the changes were actually compatible
- find out the Serial Version UID of the original class
  - use the **`serialver`** utility
- add a member variable to the changed class

```
protected static final long serialVersionUID = -2215190743590612933L;
```

- now it’s marked as compatible with the old class

## Using readObject

- if you need to force an object to be compatible
- implement readObject() method to make compatible changes

```
private void readObject(ObjectInputStream stream) throws
  java.io.IOException
{
   defaultReadObject(stream);
   // do compatible stuff
}
```

## Callbacks

- They just work
- Pass in a remote reference to a client object
- Server object can call its methods transparently
- Registry is out of the loop

## RMI Security

- Server is untrusted
- Stubs could be malicious
- rmic is OK, but someone could custom-code an evil stub: it’s just a .class file

## Limitations of RMI

- Java-only
  - but you can use JNI on the server
- Uses TCP, not UDP
- At least two sockets per connection
- Untested for huge loads

## RMI vs. COM

- Very similar
- remote interfaces ~ type libraries
- COM is Win32-only (for now)

## Sun vs. Microsoft

- RMI is not shipped as part of Microsoft’s products
- RMI will still work in applications
  - include java.rmi.* class files in your classpath
  - download rmi.zip from ftp.microsoft.com
- RMI will work in applets
  - include java.rmi.* class files (or rmi.zip) in your codebase
  - IE4: only if they’re signed
  - extra download time