CMSC 691
High Performance Distributed Systems

Java threads and RMI

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

Java threads: Runnable interface

- Provide a *Runnable* object. The Runnable interface defines a single method, *run*, meant to contain the code executed in the thread
- The Runnable object is passed to the Thread constructor
- *start* and *join* control the beginning and end of the start

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) throws Exception {
         Thread t = new Thread(new HelloRunnable());
         t.start();
         t.join();
    }
}
```

Java threads: Thread class inheritance

- The Thread class itself implements Runnable
- An application can subclass Thread, providing its own implementation of the *run* method

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) throws Exception {
         HelloThread t = new HelloThread();
         t.start();
         t.join();
    }
}
```

Thread priorities and scheduling

- Threads have priority from 1 to 10

Thread.MIN_PRIORITY	1
Thread.NORM_PRIORITY	5 (default)
Thread.MAX_PRIORITY	10

- New threads inherit priority of thread that created it

```
public void run(){
  int priority = Thread.currentThread().getPriority();
}

public static void main(String args[]) throws Exception {
  Thread t1 = new Thread();
  Thread t2 = new Thread();
  t1.setPriority(Thread.MIN_PRIORITY);
  t2.setPriority(Thread.MAX_PRIORITY);
  t1.start();
  t2.start();
}
```

Synchronized methods

- Synchronized is used to lock an object for any shared resource
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task
- Also valid for static synchronized methods

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
      c++;
    }

    public synchronized int value() {
      return c;
    }
  }
```

Synchronized statements / blocks

- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock (monitor)
- Synchronized block is better than synchronized method because by using synchronized block you can only lock critical section of code and avoid locking the whole method which can possibly degrade performance of the parallel application

```
public class SynchronizedCounter {
    private int c = 0;

    public void increment() {
      synchronized(this){
        c++;
      }
    }
}
```

Atomic access

- Package *java.util.concurrent.atomic* provides a small toolkit of classes that support lock-free thread-safe programming on variables
- Examples: AtomicInteger, AtomicReference<V>

volatile keyword

- Used to indicate that a variable's value will be modified by threads
- The value of this variable will never be cached thread-locally
- In case only one thread reads and writes the value of a volatile variable and other threads only read the variable, then the reading threads are guaranteed to see the latest value of the volatile variable
- Does **NOT** protect from race conditions (multiple threads R&W)

Locks

- *Java.util.concurrent.locks.Lock* is a thread synchronization mechanism just like synchronized blocks
- First a Lock is created. Then it's lock() method is called and the Lock instance is locked. Any other thread calling lock() will be blocked until the thread that locked the lock calls unlock(). Finally unlock() is called, and the Lock is now unlocked so other threads can lock it

```
private int c = 0;
private Lock lock = new ReentrantLock();

public void increment() {
  lock.lock();
  c++;
  lock.unlock();
}

public int value() {
  return c;
}
```

Semaphores

- A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid missed signals, or to guard a critical section like you would with a lock
- Binary or counter semaphores
- *acquire* and *release* methods control the signaling

```
private int c = 0;
private Semaphore semaphore = new Semaphore(1);

public void increment() {
  try {
    semaphore.acquire();
    c++;
    semaphore.release();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

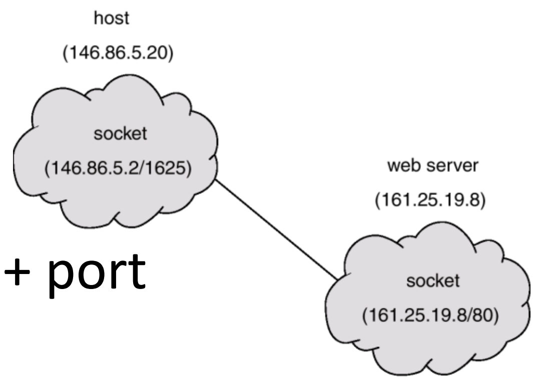
Thread Pools

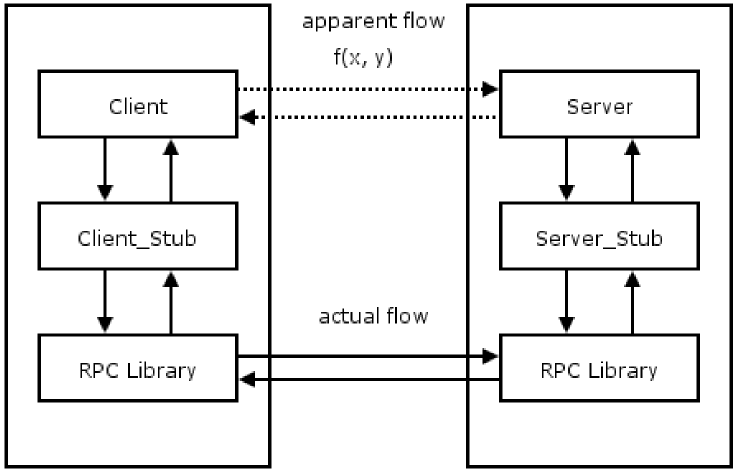
- Useful when you need to limit the number of threads running
- Instead of starting a new thread for every task to execute concurrently, the task is passed to a pool. As soon as the pool has any idle threads the task is assigned to one of them and executed.

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

Client-server communication

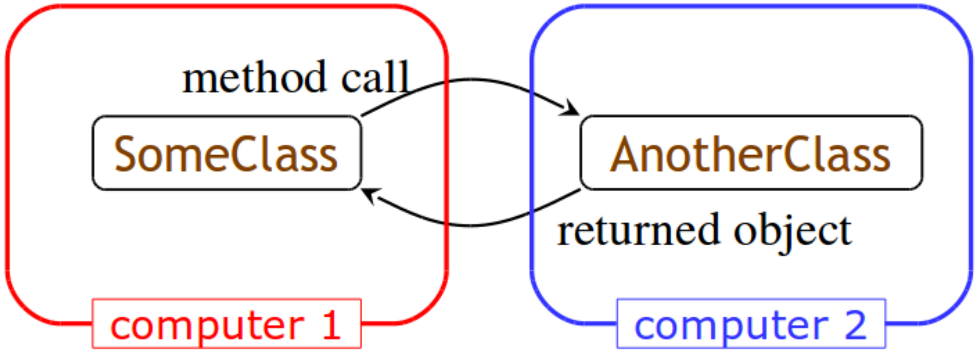
- Sockets
  - Low-level endpoint for communication: IP + port
- Remote Procedure Call (RPC)
  - Protocol to request a service from a program located in another computer in a network





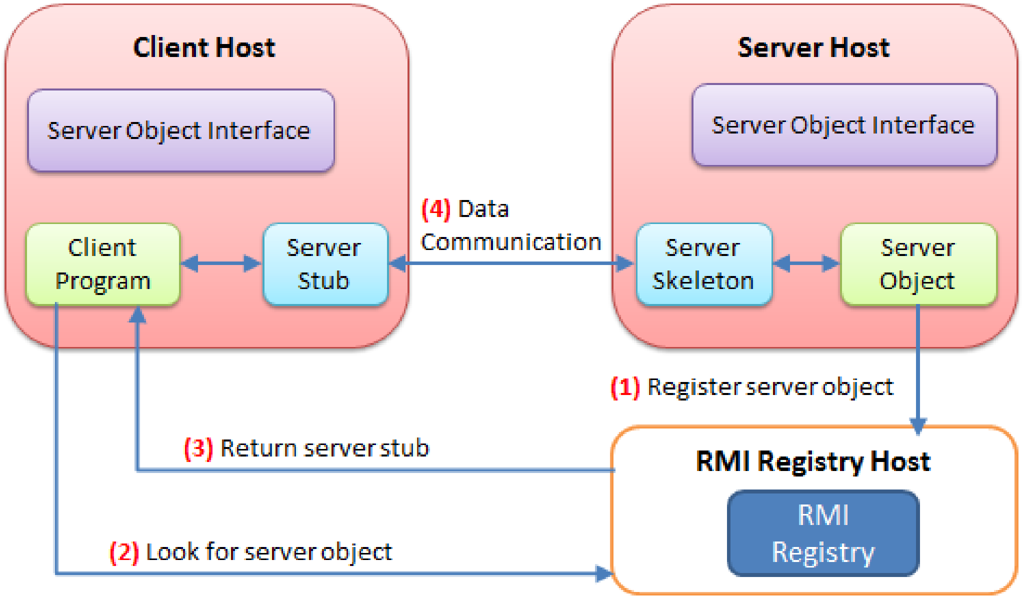
Remote method invocation technologies

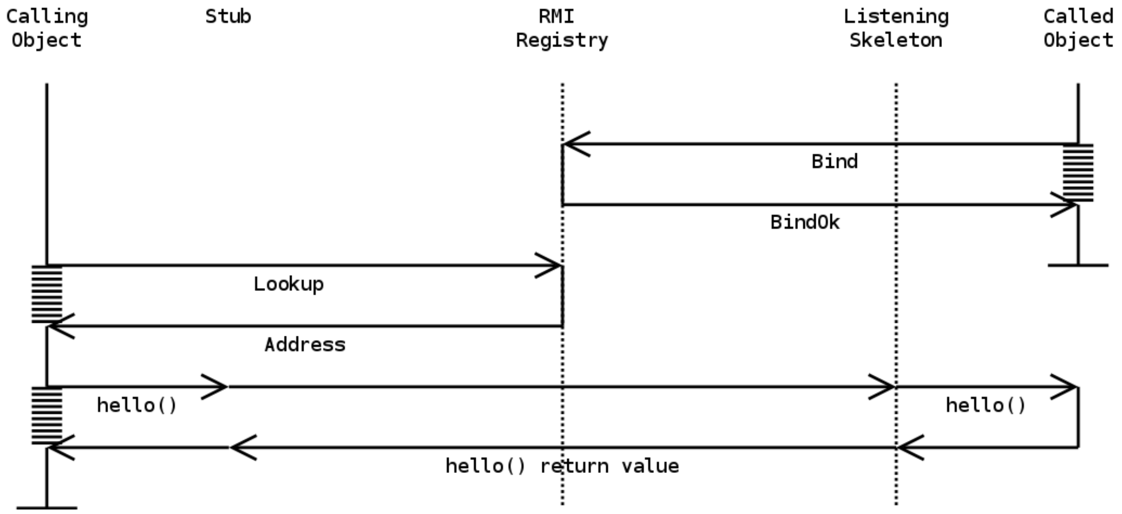
- CORBA (Common Object Request Broker Architecture)
  - CORBA supports object transmission between any languages
  - Objects described using IDL (Interface Definition Language)
  - CORBA is complex and flaky
- Java RMI (Remote Method Invocation)
  - Java to Java communications only, much simpler than CORBA



Java RMI architecture

- Client
- Server
- Server interface
- Server object
- RMI registry





Java RMI architecture

- In order to use a remote object, the client must know its behavior (interface), but does not need to know its implementation (class)
- In order to provide an object, the server must know both its interface (behavior) and its class (implementation)
- A Remote class is one whose instances can be accessed remotely
- A Serializable class is one whose instances can be marshaled (turned into a linear sequence of bits) and transmitted from one computer to another. The class must implement Serializable.

Java RMI architecture: Remove server

- You can send messages to a Remote object and get responses back
- All you need to know about the Remote object is its interface
- You can transmit a **copy** of a Serializable object between computers

- The server class should extend **UnicastRemoteObject**
- The server class needs to register its server object:

```
String url = "rmi://" + host + ":" + port + "/" + objectName;
Naming.rebind(url, serverObject);

// default port 1099
// host name should be accessible by the clients
```

Hello world using RMI

- Server interface

```
public interface HelloWorldInterface extends Remote {
   public String sayHello(String name) throws RemoteException;
}
```

- Server object

```
public class HelloWorldServer
extends UnicastRemoteObject implements HelloWorldInterface {

   public String sayHello(String name) throws RemoteException {
      return "Hello " + name;
      // String for input and return are serializable
   }
}
```

Hello world using RMI

- Server class

```
public class HelloServer {
   public static void main (String[] args) throws Exception {
     Naming.rebind("rmi://localhost/HelloWorldServer",
                                           new HelloWorldServer());
   }
}
```

- Client class

```
public class HelloClient {
   public static void main (String[] args) throws Exception {
      HelloWorldInterface hello;
      String name = "rmi://localhost/HelloWorldServer";
      hello = (HelloWorldInterface) Naming.lookup(name);
      System.out.println(hello.sayHello("Alberto"));
   }
}
```

Running the registry, server, and client

- Run the registry at the server (bin path): rmiregistry
- Run the server program at the server:

  java HelloServer -Djava.rmi.server.hostname=SERVER_IP
- Run the client program at the client: java HelloClient

Summary

1. Start the registry server, rmiregistry
2. Start the object server, the object server registers an object in the registry server
3. Start the client, the client looks up the object in the registry, makes a request
4. The Stub classes on client and server talk to each other

CMSC 691
High Performance Distributed Systems

Java threads and RMI

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu