CMSC 433 – Programming Language Technologies and Paradigms

Java RMI

Distributed Computing

- Programs that cooperate and communicate over a network
  - E-mail
  - Web server and web client
  - SETI @Home

Distributed Computing

- Machines are not all the same
  - But all adhere to same communication protocol
- Network is "slow"
  - Sending a message takes a lot of time
- Network is unreliable
  - Machines may join and leave with no warning
  - Part of the network may fail

Distributing Computations

- Connecting via sockets
  - e.g., Logging Server examples
  - Custom protocols for each application
- RPC/DCOM/CORBA/RMI
  - Make what looks like a normal function call
  - Function actually invoked on another machine
  - Arguments/return values are marshalled / unmarshalled for transport across the network

Remote Method Invocation

- Goal - easy way to get distributed computation
- Create proxies for remote objects
  - Calls to proxy get translated into network calls
  - Implemented on top of sockets
- Arguments and return values are passed over network
  - Java takes care of the details

A Simplified Example

```
// runs on one mach.
class ChatServerImpl implements ChatServer ... {
  public void say(String s) {
    System.out.println(s);
  }
  ...
}
class Chatter { // runs on another mach.
  public static void main(String args[]) {
    ChatServer c = // get remote object;
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    while (true) {
      System.out.print("> ");
      c.say(br.readLine());
    }
} }
```

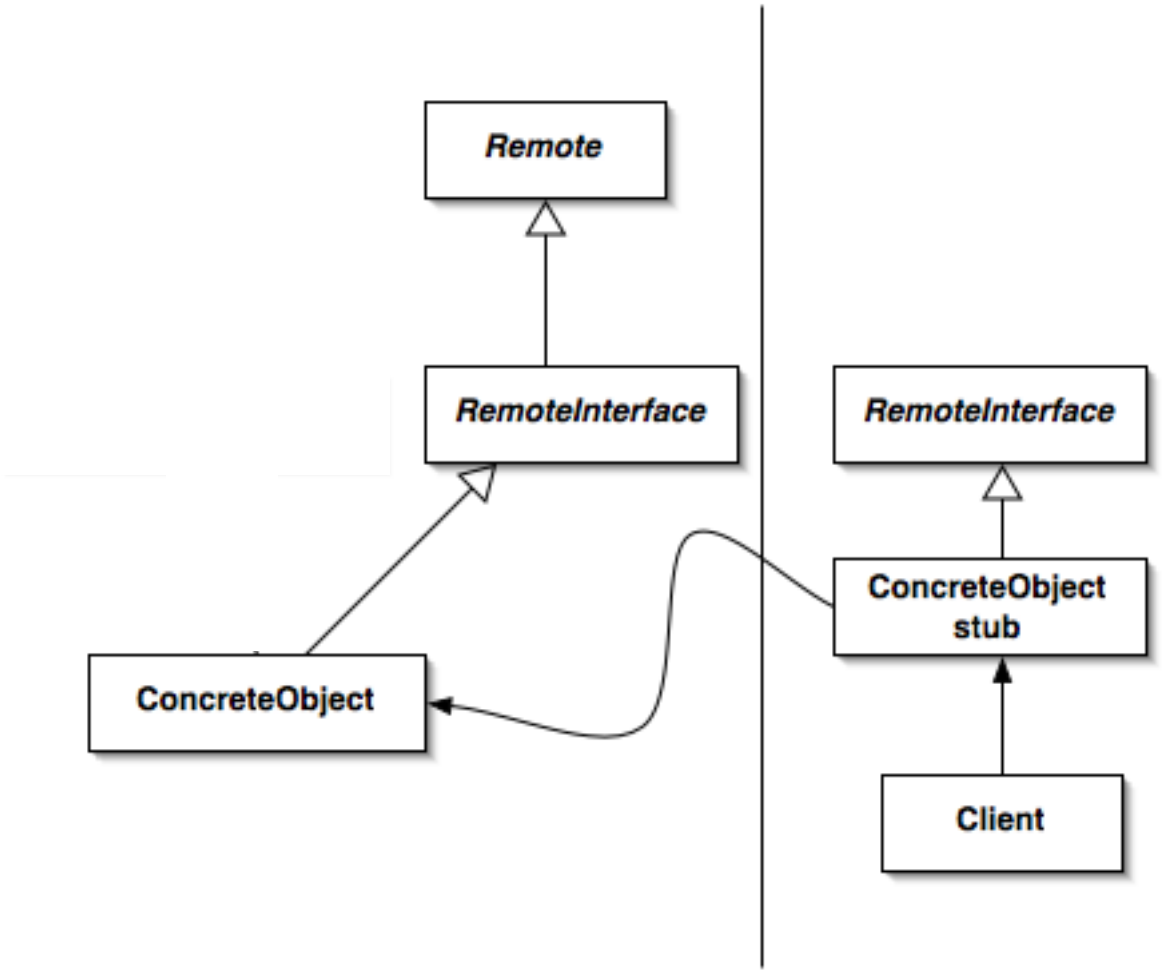
Remote Objects

- Remote Objects implement a Remote interface
- A Remote interface extends java.rmi.Remote
- All Remote interface methods throw RemoteException
- Constructor throws RemoteException
  - RemoteException means “something bad happened on the network”

Stubs

- Client only sees the RemoteInterface
  - ConcreteObject can have other methods
- Remote Objects represented using stub
  - Stub sends arguments over network
  - Stub receives result back from network

Remote Interfaces



Passing Arguments

- To pass an argument to a Remote method or return a result from a Remote method, object/value must be either
  - A primitive type (int, double, etc.),
  - Serializable (e.g., String), or
  - Remote (i.e., implement a sub-interface of Remote)
- Primitives passed by value

Passing Serializable vs. Remote

- Serializable objects passed by value
  - Same Serializable object in different calls materializes different objects at receiver
- Remote objects passed by reference
  - Same Remote object in different calls yields same stub object, which passes arguments back to same remote object

Stub Code

- Classes contain both data and code
  - When you receive a Remote object, you need the stub for that object
- Where does it come from?
- Solution #1: Make all clients have the stub code on their classpath
  - Or stub code for another class with same remote interface

Downloading Code

- Solution #2: Provide a *codebase* where stub code for objects can be downloaded

  java -Djava.rmi.server.codebase=<url> ...

  – Specifies location of code for classes that originate **in this JVM**

  – URL - can be http://, file:/, etc.

Getting the First Remote Object

- Can publish objects to an RMI registry
  - Each object has a name (that you specify)
  - Registry listens on a port (1099 default)
- Naming.lookup(url) gets object from registry
  - e.g., Naming.lookup("rmi://localhost/Chat");
  - Used to get first reference to Remote object
  - Don't need to lookup objects returned by Remote methods

Starting an RMI Registry

- Method 1: Separate RMI registry process
  - Command `rmiregistry`
    - Run with stubs in classpath, or specify codebase
  - Listens on port 1099 by default
  - Pros: Registry doesn't die when your program dies
    - Multiple applications can share registry
- Method 2: Start registry in same JVM
  - LocateRegistry.createRegistry(int port)
  - Pros: Registry dies when your program dies
    - Especially useful during testing
    - No registries lying around on machine

Exporting the Remote Object

- UnicastRemoteObject.exportObject(Remote, int) exports (activates) the Remote object so that it can receive invocations of its remote methods from Remote clients
- The second argument specifies which TCP port to listen on for incoming remote invocation requests for the object.
  - The value zero specifies the use of an anonymous port
  - Use anonymous ports for your class projects
  - In practice, might use a different port to avoid firewalled ports
- Method returns a stub for the exported Remote object

Advertising Remote Objects

- Call Naming.{bind/unbind/rebind} to manipulate objects in registry
  - E.g., Naming.bind("rmi://localhost/Chat");
- Can bind/unbind/rebind name only on localhost
- Can lookup name on any host

Example: RMI Chat Server

- Server
  - Runs the chat room
- Client
  - Participant in chat room
  - Receives messages from others in room
- Connection
  - Links client to Server
  - Used to speak in chat room

Server

```
interface Server extends Remote {
   Connection logon(String name, Client c)
        throws RemoteException;

   public Map<String,Client> getUsers()
             throws RemoteException;

   }
```

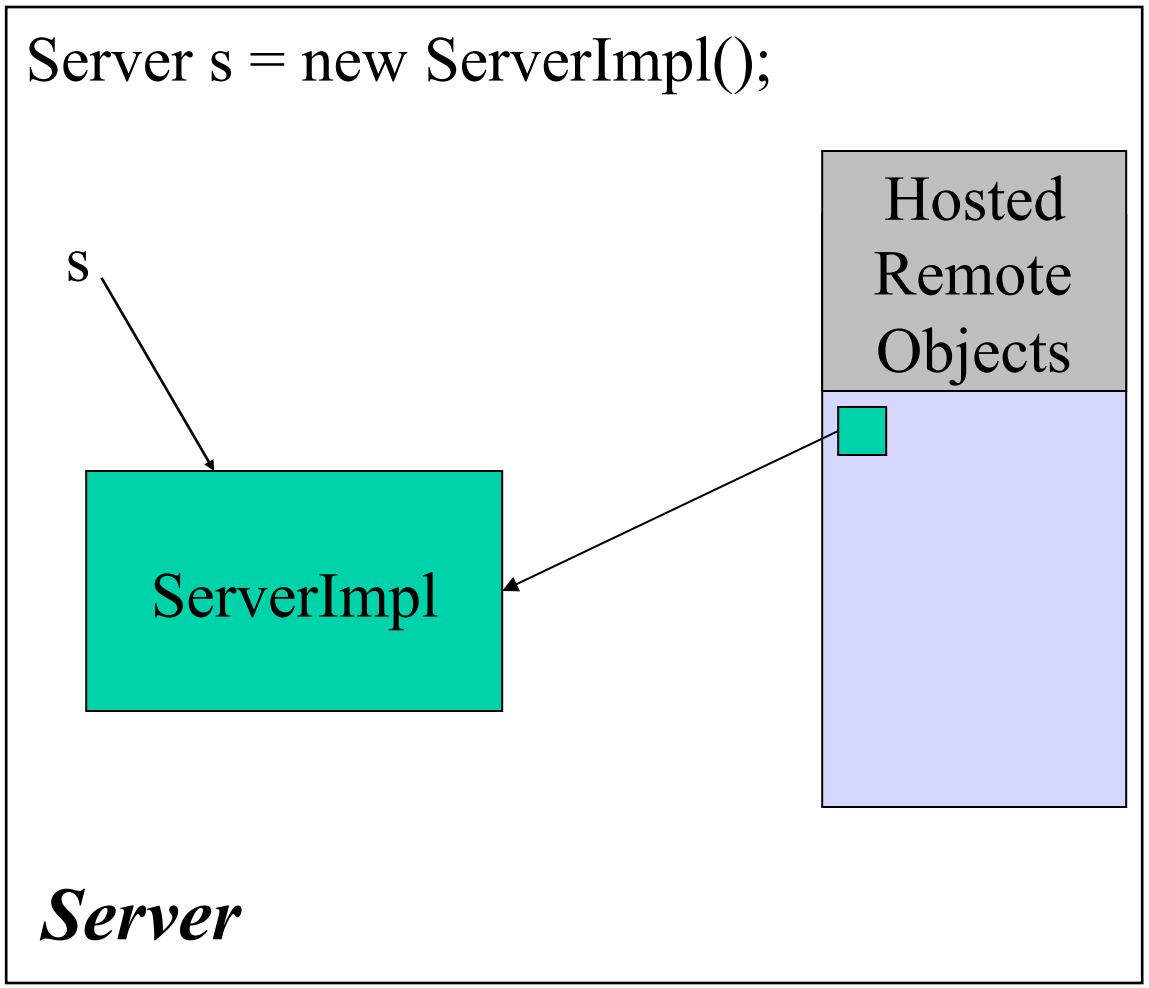
Connection

```
interface Connection extends Remote {

        /** Say to everyone */
        void say(String msg)
                throws RemoteException;

        / ** Say to one person */
        void say(String who, String msg)
                throws RemoteException;

        String [] getUsers()
                throws RemoteException;

        void logoff()
                throws RemoteException;
        }
```

Client

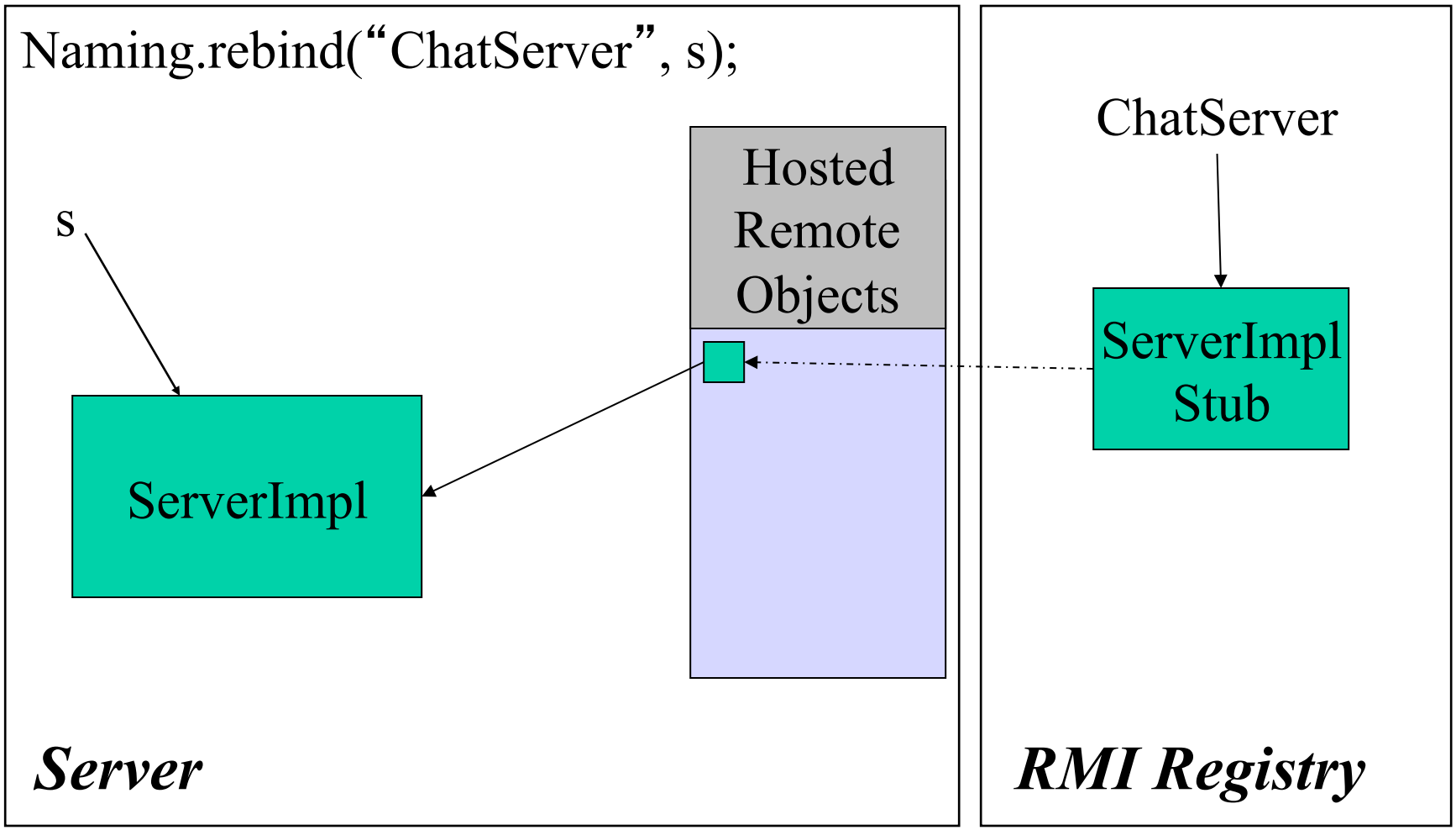
```
interface Client extends Remote {

        void wasSaid(String who, String msg)
                throws RemoteException;

        void usersChanged(String [] who)
                throws RemoteException;
        }
```

Server's Remote Object Creation

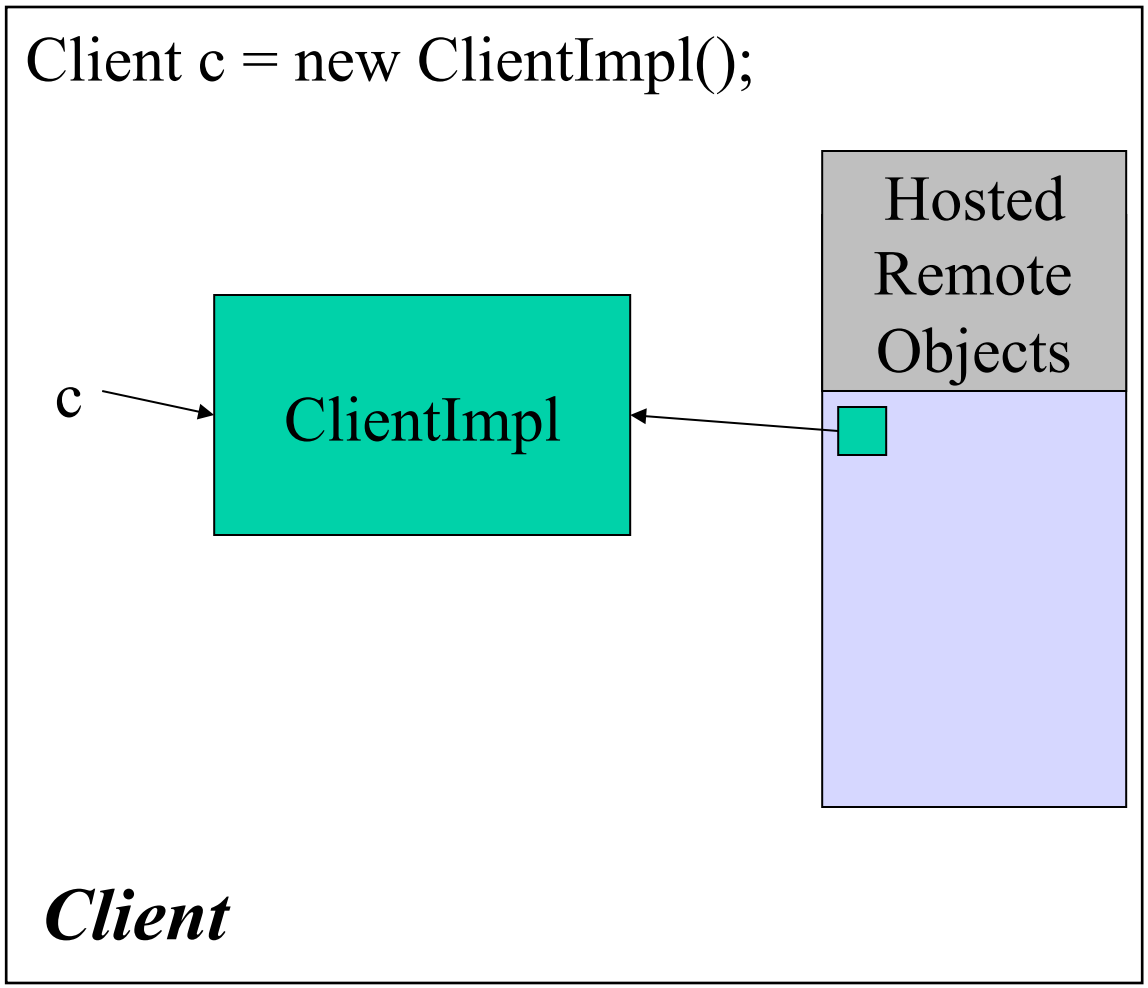


*Object added to table because it implements extension of **Remote** interface*

Remote Object Registry

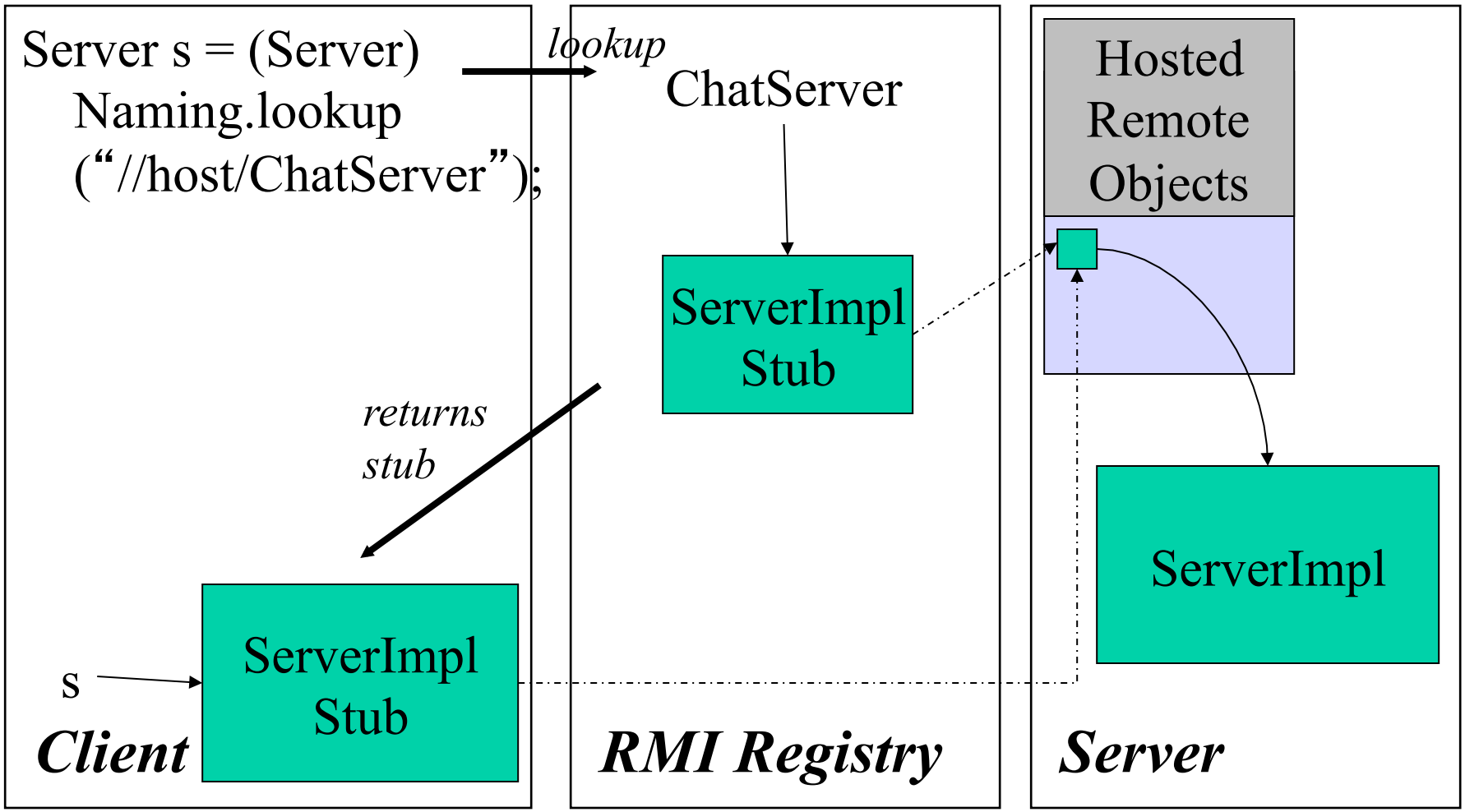


Client's Remote Object Creation

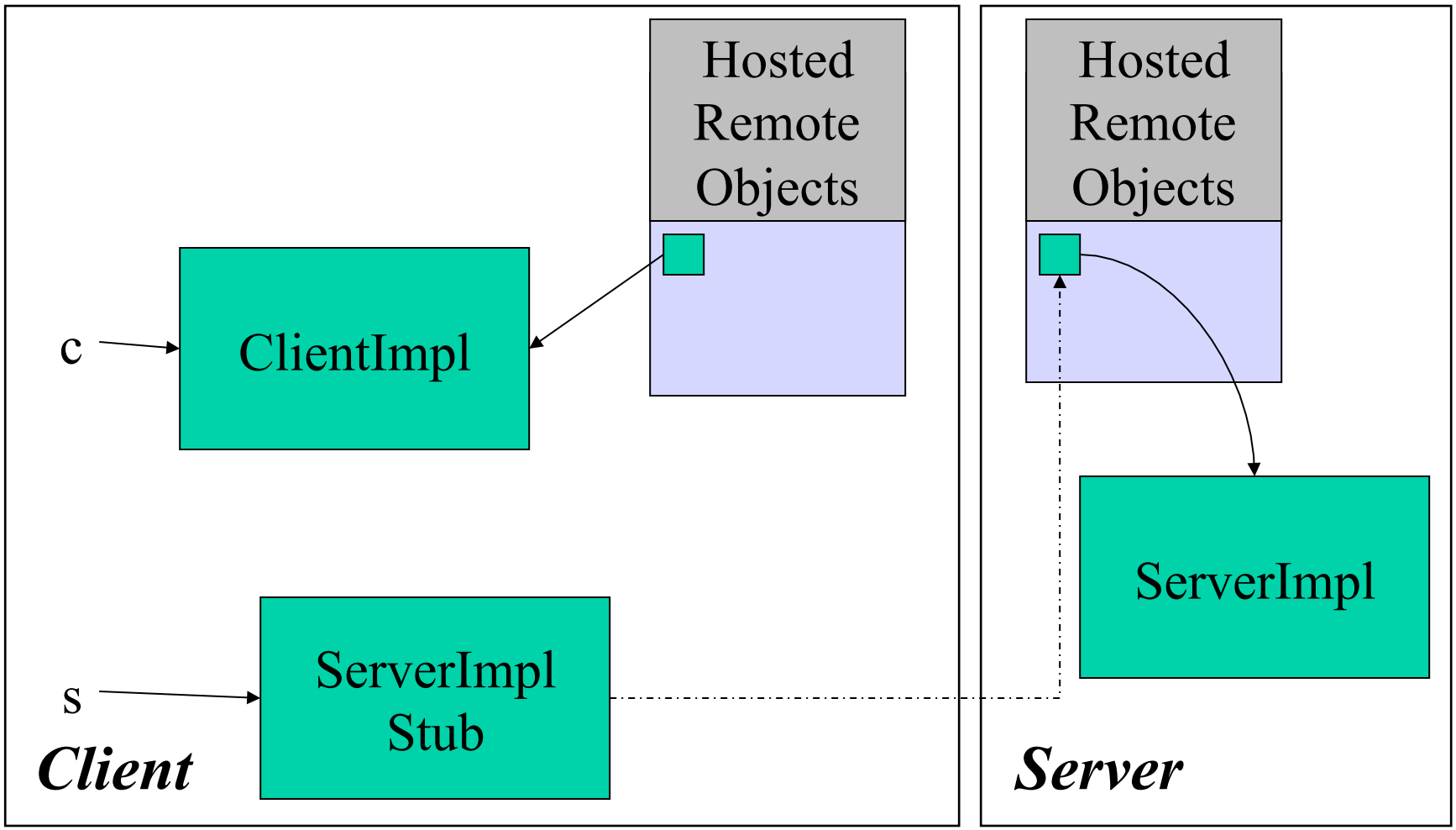


*Client object also implements extension of **Remote** interface*

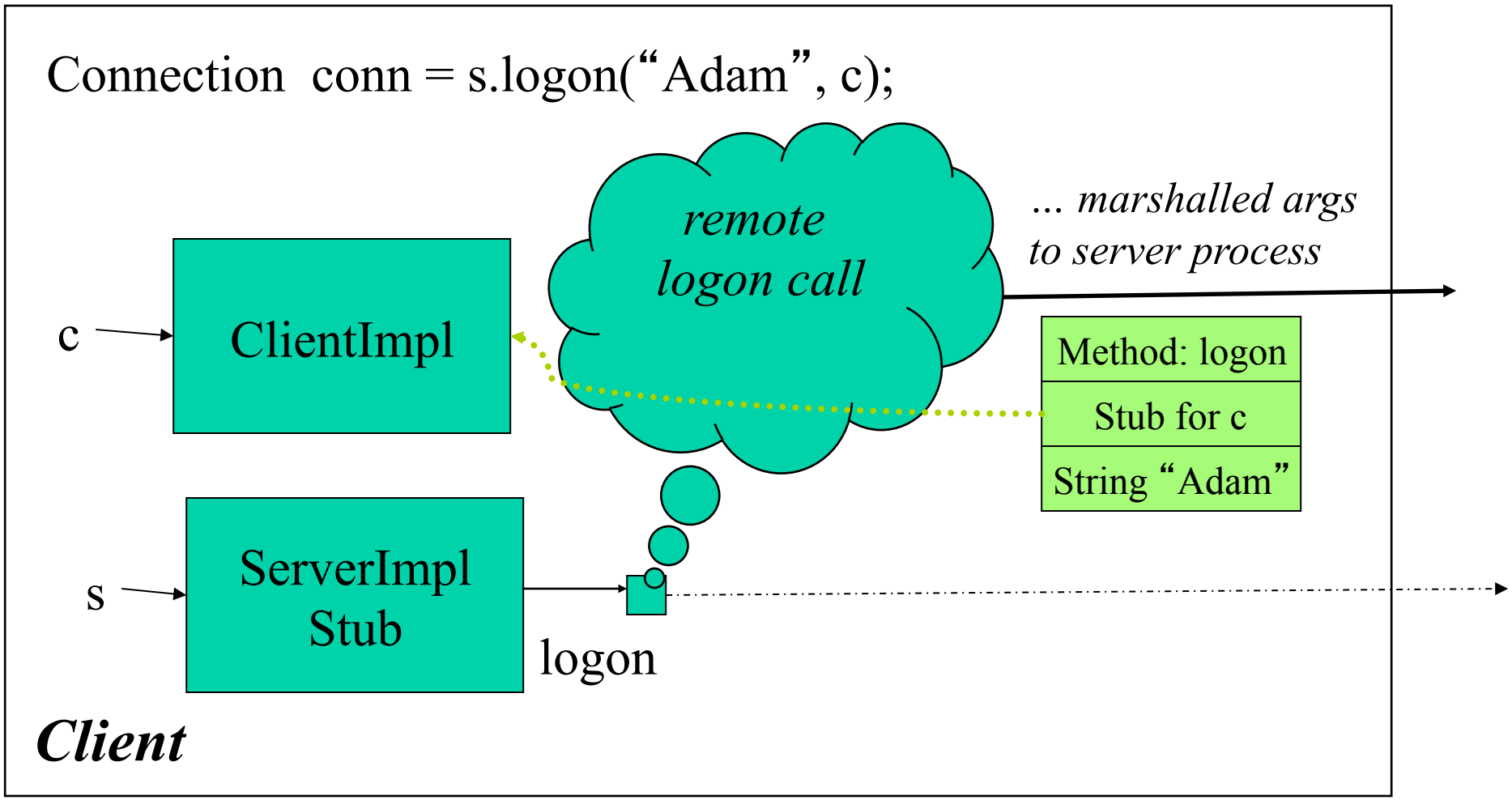
Client Looks Up Server



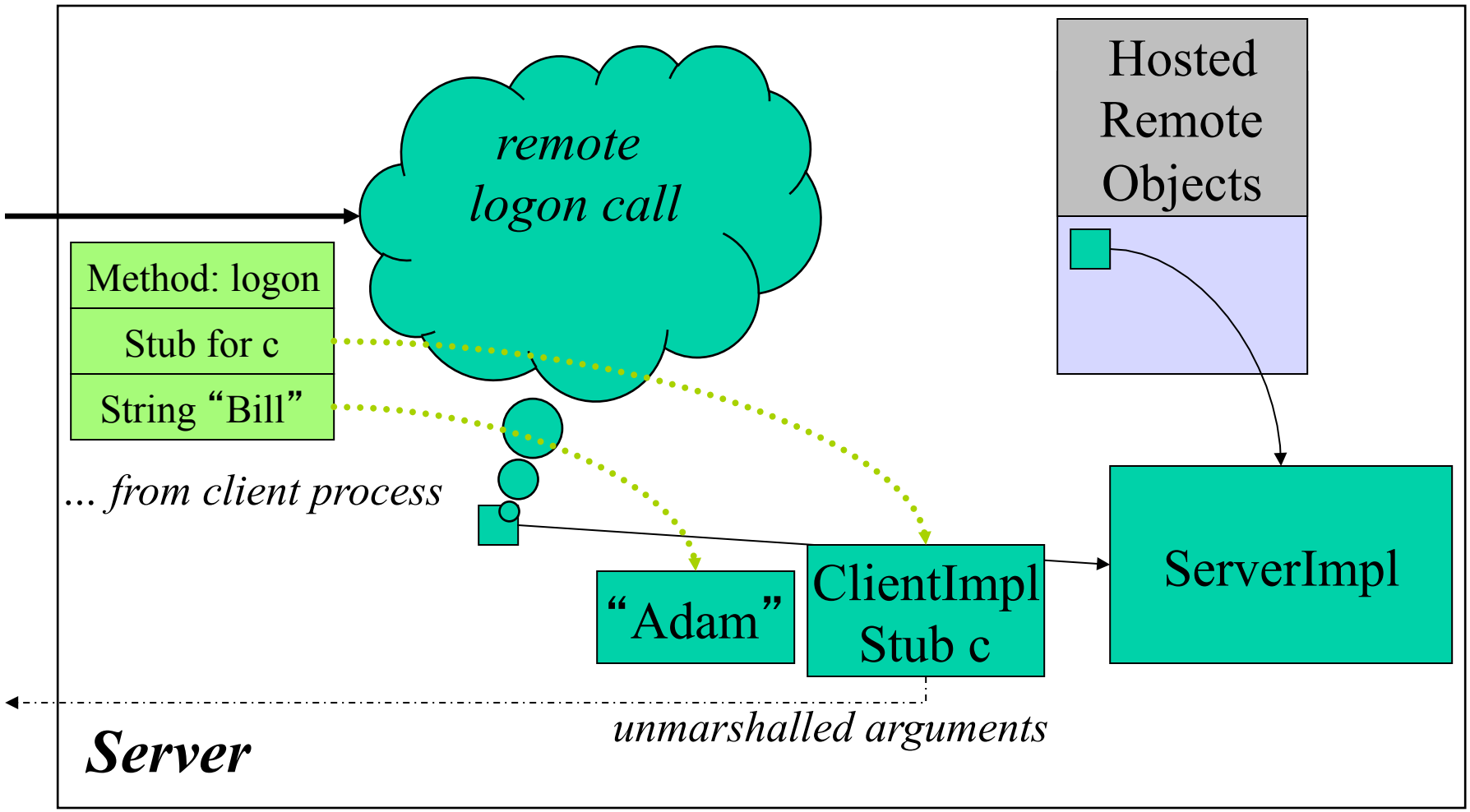
After Lookup Finished



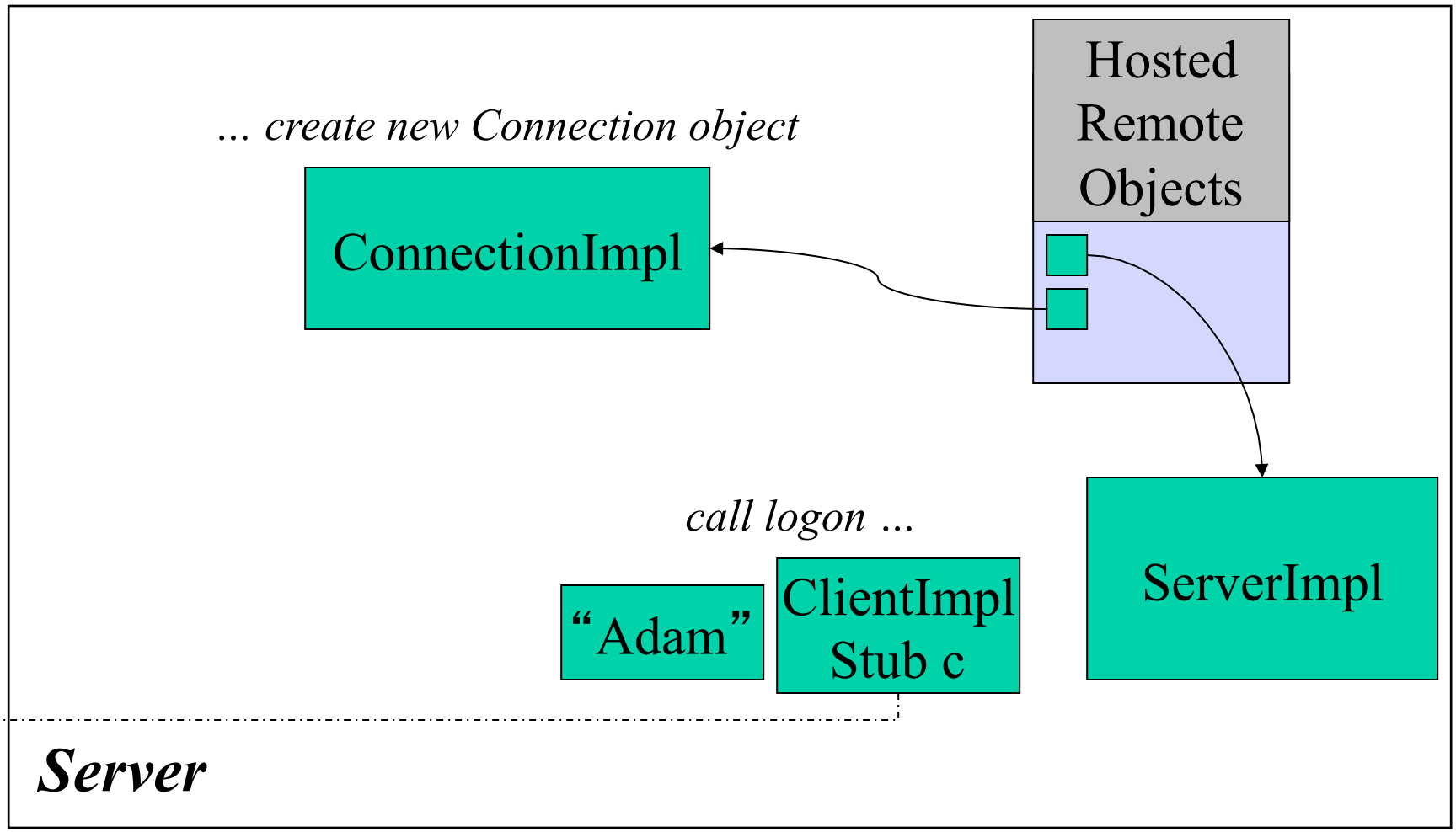
Client Invokes Remote Method



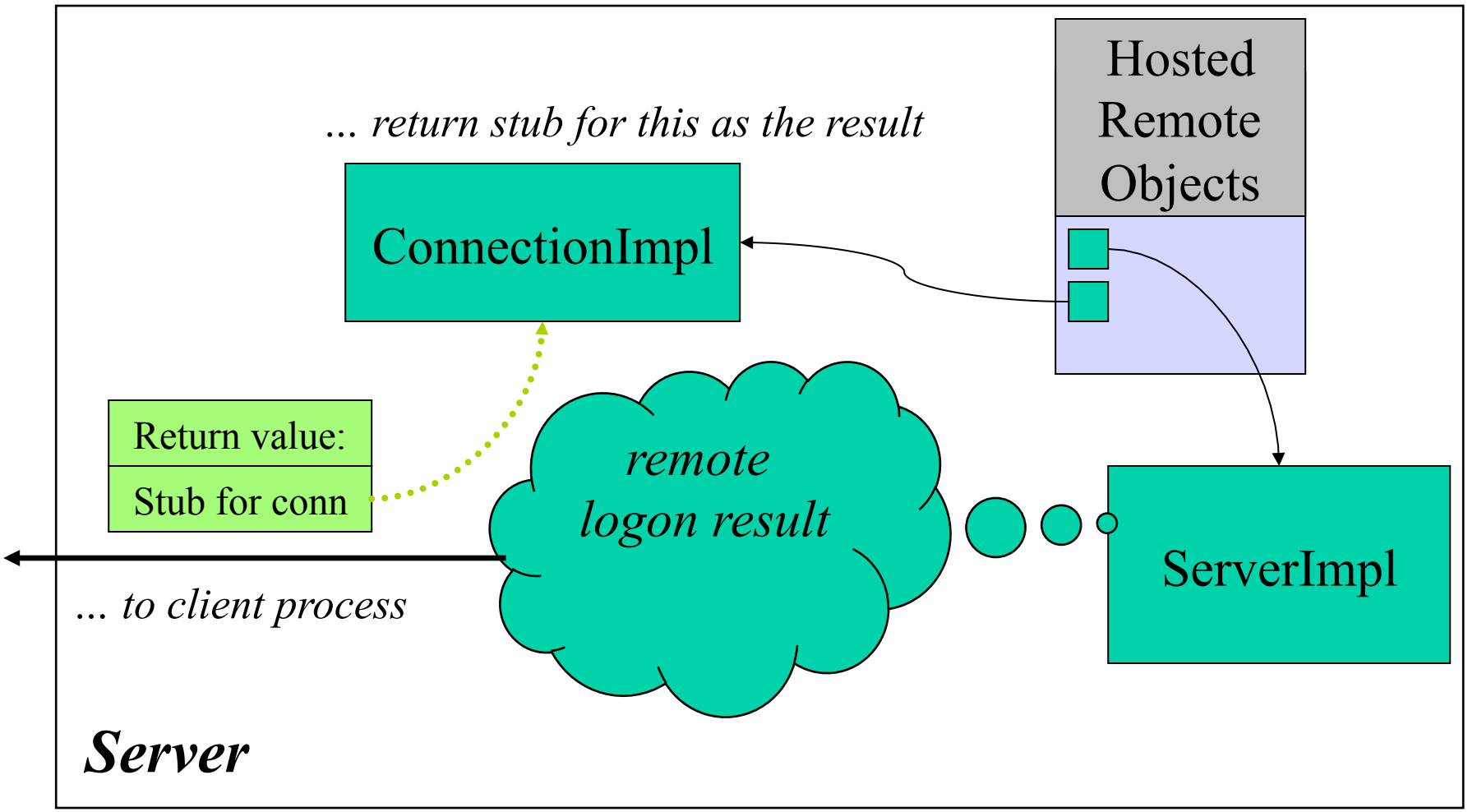
Server Receives Remote Call



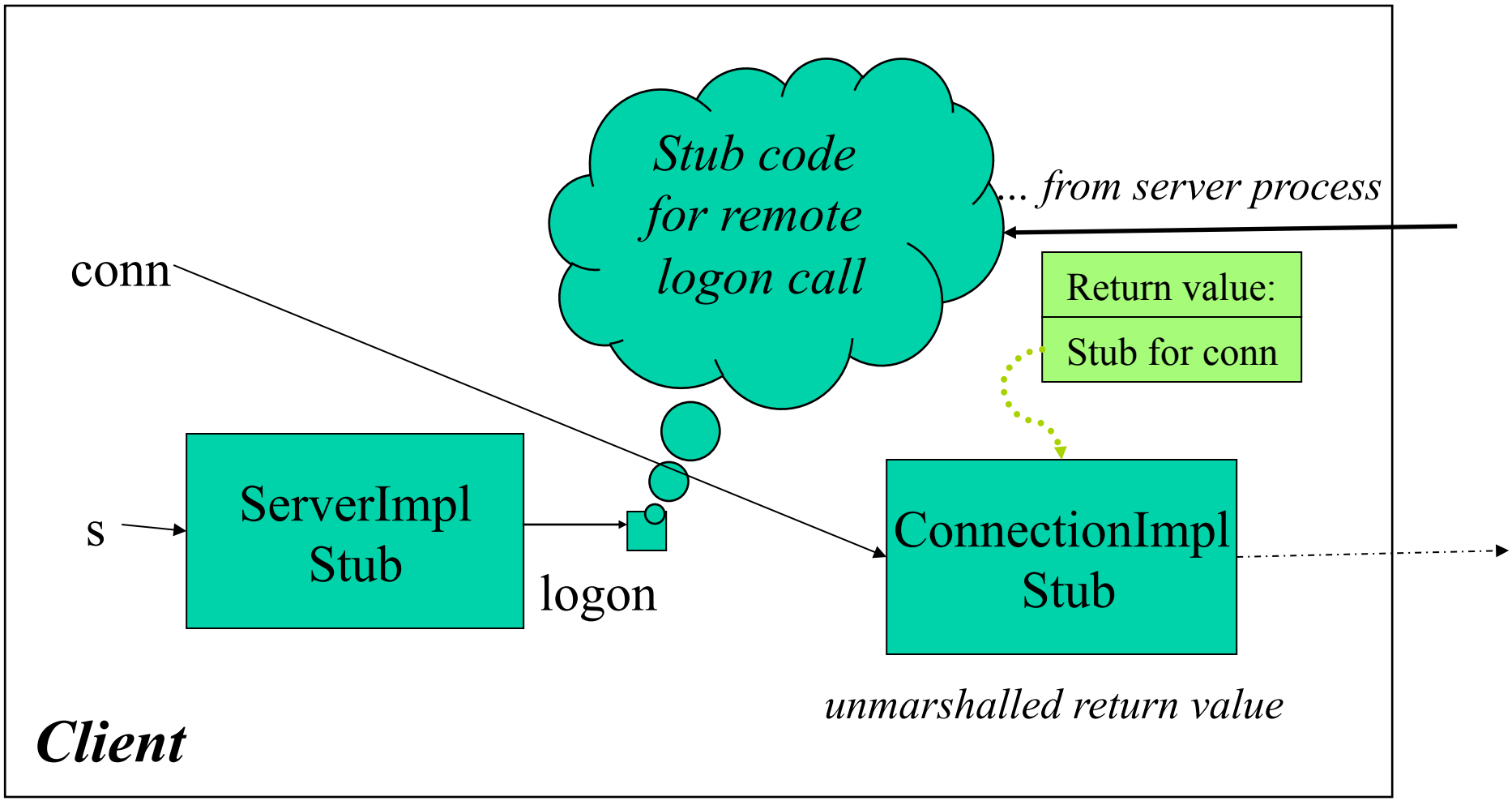
Server Executes the Call



Server Returns the Result



Client Receives the Result



Security Manager

- When using a codebase, we must download stub code from a remote site. This is potentially risky
  - Need to limit what downloaded code could do
  - Must install a Security Manager before you download any code from RMI codebases
- Can use

```
System.setSecurityManager(
                    new RMISecurityManager());
```

Policy Files

- In addition to security manager, need to specify a security policy, e.g.,

```
grant {
  permission java.net.SocketPermission "*:
  1024-65535", "connect,accept";
  permission java.net.SocketPermission "*:80",
  "connect";
};
```

- Set security policy when JVM started
  - java -Djava.security.policy==<file name>
  - Note above: behavior when using "==" is different from just using "="

Debugging Tips

- See:
  - http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/logging.html
- -Djava.rmi.server.logCalls=true
  -Dsun.rmi.server.logLevel=VERBOSE
  -Dsun.rmi.loader.logLevel=VERBOSE