G52CON: Concepts of Concurrency

Lecture 16 Distributed Processing in Java II

Natasha Alechina

School of Computer Science

nza@cs.nott.ac.uk

Outline of this lecture

- Remote invocation and `java.rmi`
- RMI applications
- RMI mechanics
- example:
  - remote `RWDictionary`

Synchronising communication

If a process tries to **`receive`** a message before one has been sent, it will block until there is a message for it to read.

The differences are mainly in the behaviour of the **`send`**ing process:

- asynchronous communication: the sending process continues without waiting for the message to be received, e.g., Unix sockets, `java.net`
- synchronous communication: the sending process is delayed until the corresponding receive is executed, e.g., CSP, occam
- remote invocation: the sending process is delayed until a reply is received, e.g., RPC (`java.rmi`), Extended Rendezvous

Remote invocation

- remote invocation combines aspects of *monitors* and *synchronous message passing*:
  - as with monitors interaction is via public procedures; and
  - as with synchronous send, calling a procedure delays the caller.
- a remote invocation is a *two way* communication channel from the caller to the process servicing the call and back.
- remote invocation is implemented using message passing

Modules

A module contains both processes and local and exported procedures:

- the *header* contains the signatures of the exported procedures
- the *body* contains local procedures and processes, local variables, and initialisation code
- at any point in time, a module contains zero or more *processes*
- different modules may reside in different addresses spaces

Modules and message passing

Communication *between* modules is by calls to exported procedures:

- arguments and return values are passed as *messages*
- the sending and receiving of messages is *implicit* rather than explicitly programmed

Communication *within* modules is similar to monitors:

- processes within a module can share variables and call procedures declared in that module.

RPC & Extended Rendezvous

There are two main forms of remote invocation:

- *Remote Procedure Call* creates a *new* process to handle each call
- *Extended Rendezvous* services a request using an *existing* process.

Modules and RPC

In RPC, a module contains *zero or more* processes and some exported procedures:

- local processes are called *background processes*
- processes that result from remote calls to exported procedures which are called *server processes*

`java.rmi`

The package `java.rmi` implements Java's version of RPC:

- remote invocation is based on the model of a *procedure call*
- in Java, non-static methods must be invoked on an *object*
- Java therefore requires both *remote methods* (procedures) and *remote objects* on which the remote methods can be invoked.

Java remote objects

A Java *remote object* is one whose methods can be invoked from another JVM, potentially on a different host:

- a remote object is described by one or more *remote interfaces* which extend `java.rmi.Remote`
- *remote method invocation* (RMI) is the action of invoking a method of a remote interface on a remote object.

Modules and `java.rmi`

- a `Remote` interface is similar to the *header* of a module containing the signatures of the exported procedures
- a class implementing a `Remote` interface is similar to the the *body* of the module, containing local methods and variables, and initialisation code
- the *processes* in a module are the threads running on the target JVM
- details of communication between `Remote` objects are handled by RMI, which uses `Sockets` and `Serialization` to implement the transfer of arguments and results

An RMI application

- a *server* application creates some remote objects, makes references to them accessible and waits for clients to invoke (remote) methods on the remote objects

- a *client* gets a remote reference to a remote object in the server, either from the RMI registry or as a return value to a remote method, and invokes (remote) methods on it

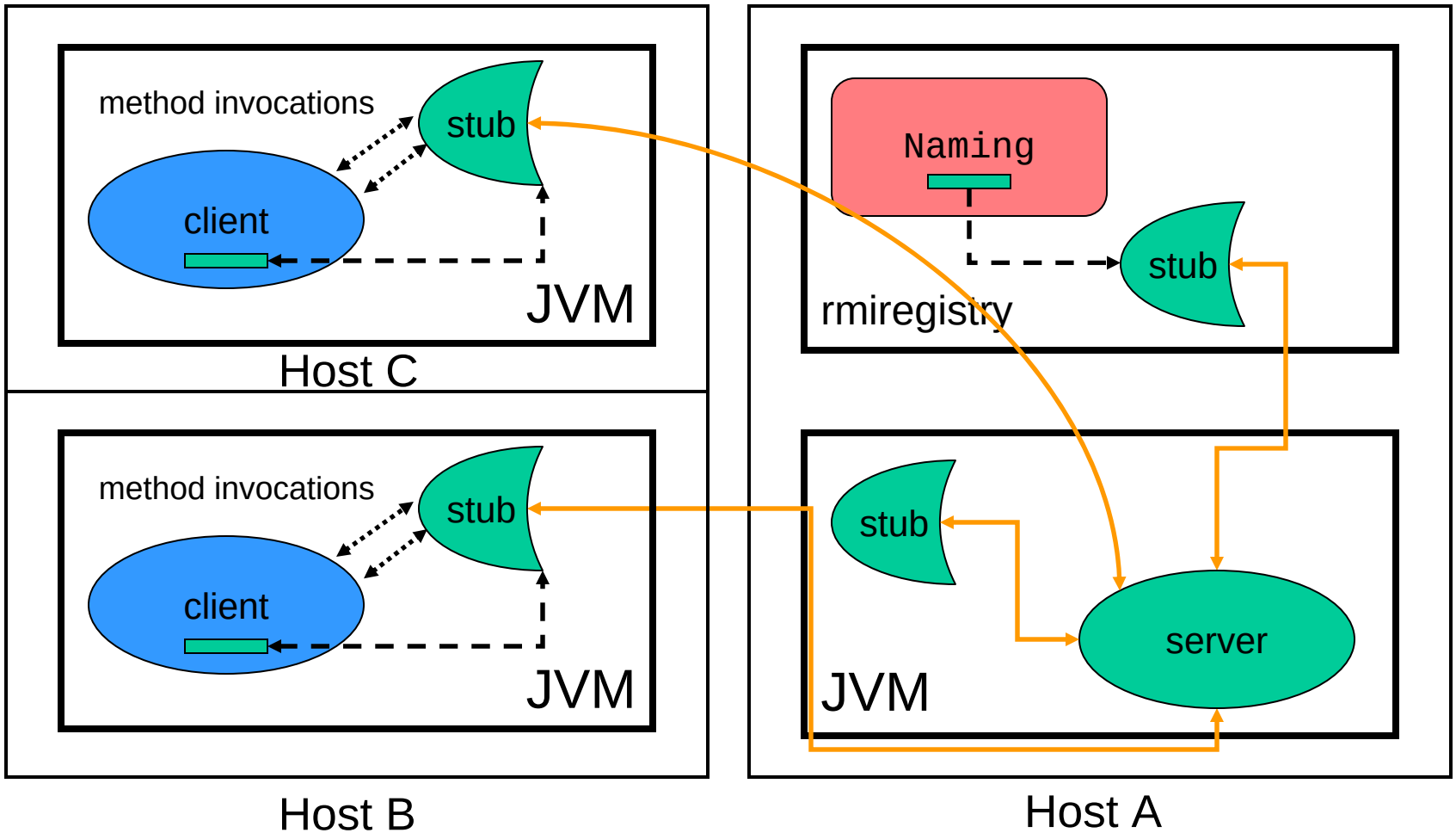
- a component of a distributed Java application can act as both a client and server

The RMI registry

The system provides a particular remote object, the *RMI registry* for finding references to remote objects:

- once a remote object is registered with the RMI registry on the local host, clients on any host can look up the remote object by name, obtain a reference to it (stub), and then invoke its methods
- the registry is typically used only to locate the *first* remote object that a client needs to use from a particular server
- the registry listens on a known port, usually 1099 on the same host as the server.

Structure of an RMI application



Stubs

A *stub* acts as a proxy for a remote object and is responsible for carrying out method calls on the remote object.

Invoking a stub method:

- initiates a connection with the remote JVM containing the remote object;
- writes and transmits the method parameters to the remote JVM;
- waits for the results of the method invocation; and
- reads the result (return value or exception) and returns it to the caller.

Parameter passing in RMI

An argument to or return value form a remote object can be any Java object that is *serializable*:

- non-remote method arguments and results are passed by *copying*– changes made to the object are not visible to other clients.

- remote objects are passed by *reference* (i.e., a copy of the stub is passed or returned)–changes made by one client to the state of the remote object are visible to all clients.

RMIC

Stubs are generated by the Java RMI Stub compiler, *rmic*

- the *rmic* command takes one or more class names as input and produces as output class files of the form `ClassName_Stub.class`

Example RWDictionary

```
class RWDictionary {
    private final Map<String, Data> m =
      new TreeMap<String, Data>();

    // locks
    private final ReentrantReadWriteLock rwl =
      new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    // methods follow ...
```

Example `RWDictionary` readers

```
// Reader method (does not update the map)
public Data get(String key) {
    r.lock();
    try {
        return m.get(key);
    }
    finally {
        r.unlock();
    }
}
```

Example `RWDictionary` writers

```
    // Writer method (changes the map)
    public Data put(String key, Data value) {
        w.lock();
        try {
            return m.put(key, value);
        }
        finally {
            w.unlock();
        }
    }
}
```

Remote `RWDictionaryServer`

```
import java.rmi.*;
import java.rmi.server.*;

public interface RWDictionaryServer extends Remote {

    public Data get(String key) throws RemoteException;

    public Data put(String key, Data value) throws
      RemoteException, ;
}
```

RWDictionaryServerImpl

```
class RWDictionaryServerImpl
  extends UnicastRemoteObject implements RWDictionaryServer {

    private final Map<String, Data> m =
      new TreeMap<String, Data>();

    private final ReentrantReadWriteLock rwl =
      new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public RWDictionaryServerImpl() throws RemoteException { }

    // continued ...
```

RWDictionaryServerImpl 2

```
// Reader method (does not update the map)
public Data get(String key)
  throws RemoteException {
      r.lock();
      try {
          return m.get(key);
      }
      finally {
          r.unlock();
      }
 }
```

RWDictionaryServerImpl 3

```
    // Writer method (changes the map)
    public Data put(String key, Data value)
     throws RemoteException {
        w.lock();
        try {
            return m.put(key, value);
        }
        finally {
            w.unlock();
        }
    }
}
```

RWDictionaryServerImpl 4

```
    public static void main(String[] args) {
        try {
            RWDictionaryServer server =
              new RWDictionaryServerImpl();
            Naming.bind("//host:port/rwDictionary", server);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

RWDictionaryServerImpl 4

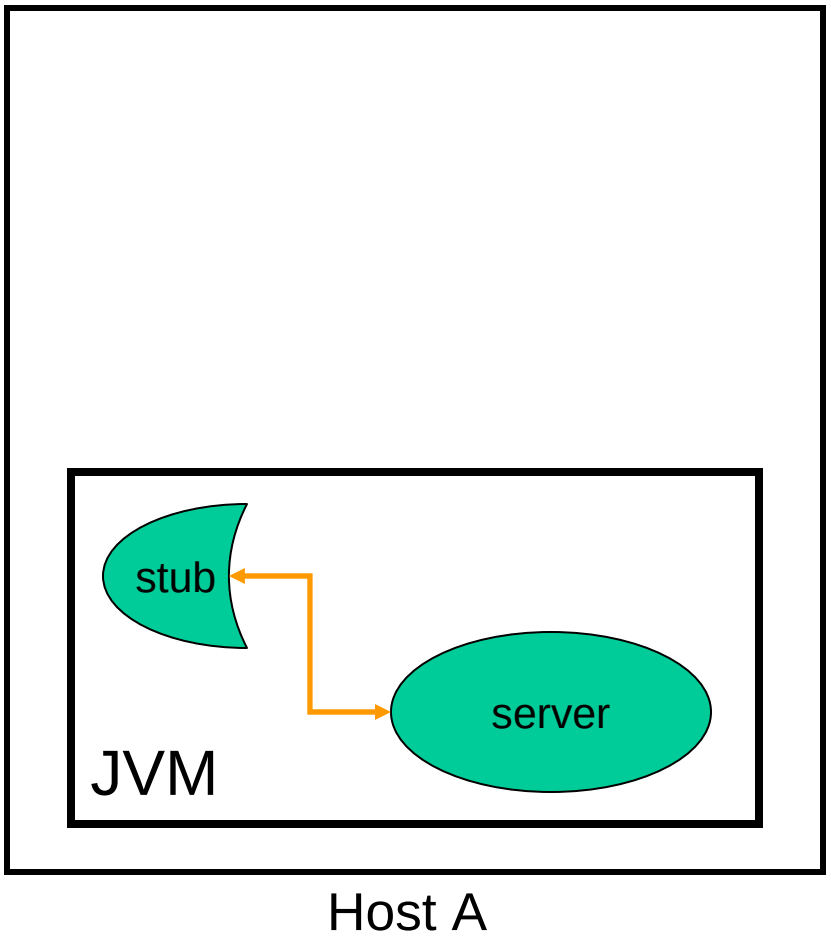
```
    public static void main(String[] args) {
        try {
            RWDictionaryServer server =
              new RWDictionaryServerImpl();
            Naming.bind("//host:port/rwDictionary", server);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

`RWDictionaryServerImpl.main()`

The main method creates an instance of the `RWDictionaryServerImpl` class

- this calls the `UnicastRemoteObject` constructor which in turn exports the newly created object to the RMI runtime
- the `RemoteRWDictionary` remote object is then ready to accept incoming calls from clients on an anonymous port chosen by RMI or the underlying OS

Exporting the server



RWDictionaryServerImpl 4

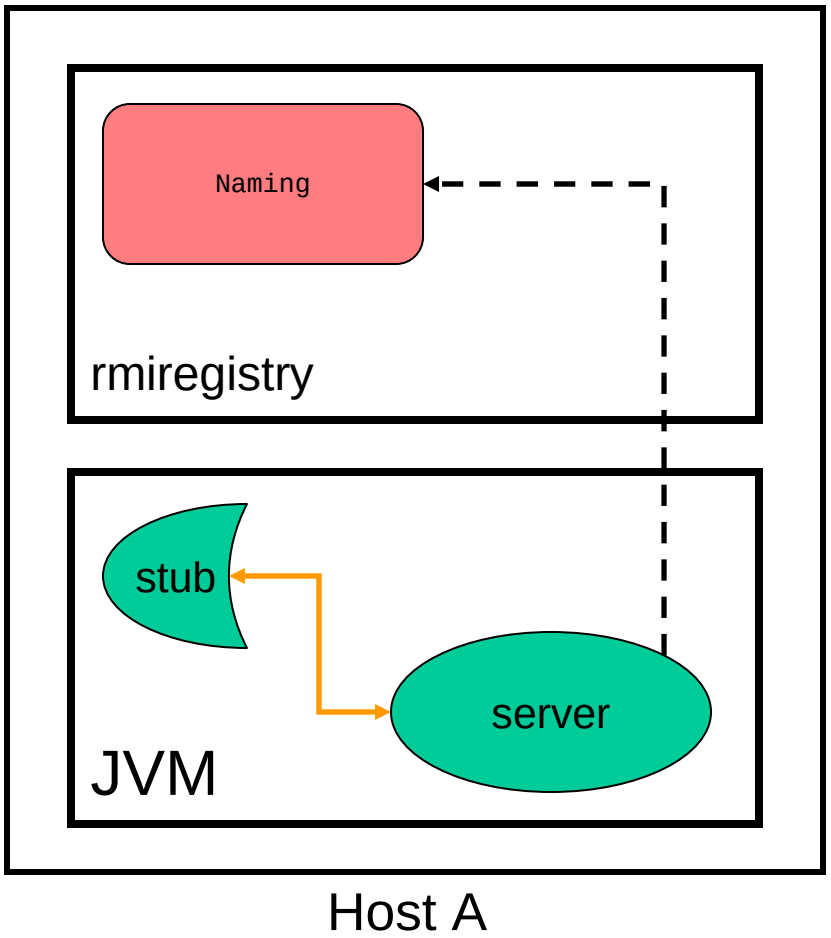
```
    public static void main(String[] args) {
        try {
            RWDictionaryServer server =
              new RWDictionaryServerImpl();
            Naming.bind("//host:port/rwDictionary", server);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

`RWDictionaryServerImpl.main()2`

Before a caller can invoke a method on a remote object, it must obtain a remote reference to it:

- the `Naming` interface is used for registering and looking up remote objects in the *registry*
- once a remote object is registered with the RMI registry on the local host, clients on any host can look up the remote object by name, obtain its reference and then invoke its methods.
- the main method then exits—as long as there is a reference to the `RWDictionaryServer` object in another JVM, on the same or a different host, the JVM will not be shut down

Naming.bind

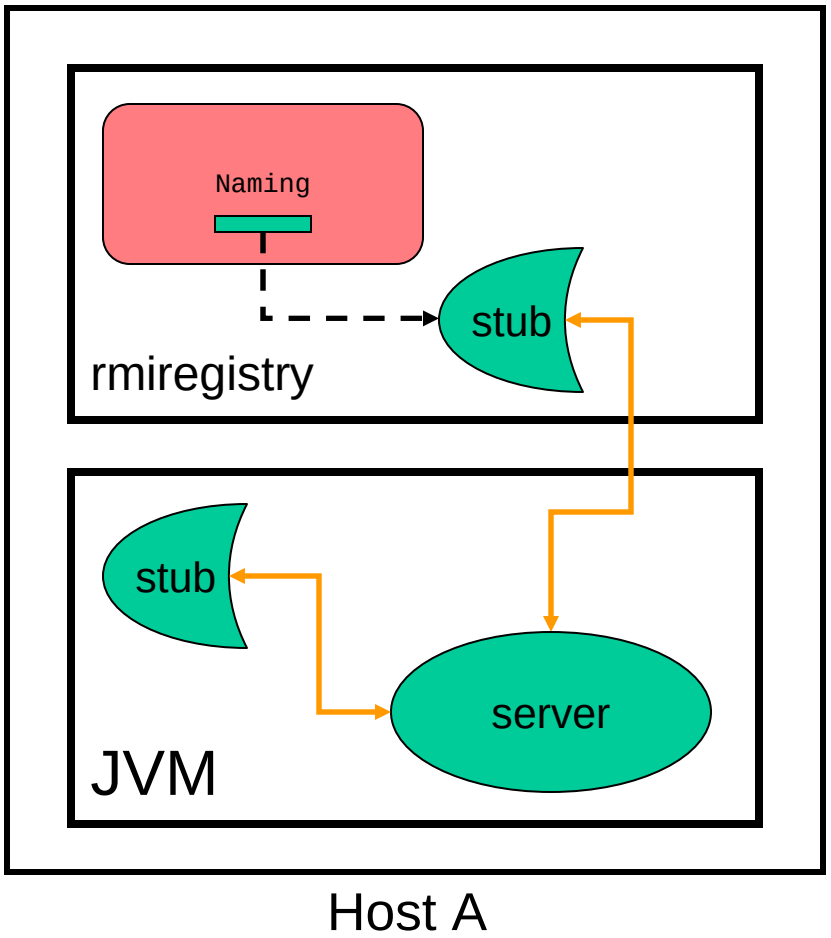


Downloading the stub

When the `RWDictionaryServerImpl` binds its remote object reference in the registry

- the registry downloads the `RWDictionaryServerImpl_Stub` and the `RWDictionaryServer` interface which the stub implements
- these come from the `RWDictionaryServerImpl`'s web server or file system

Downloading the stub



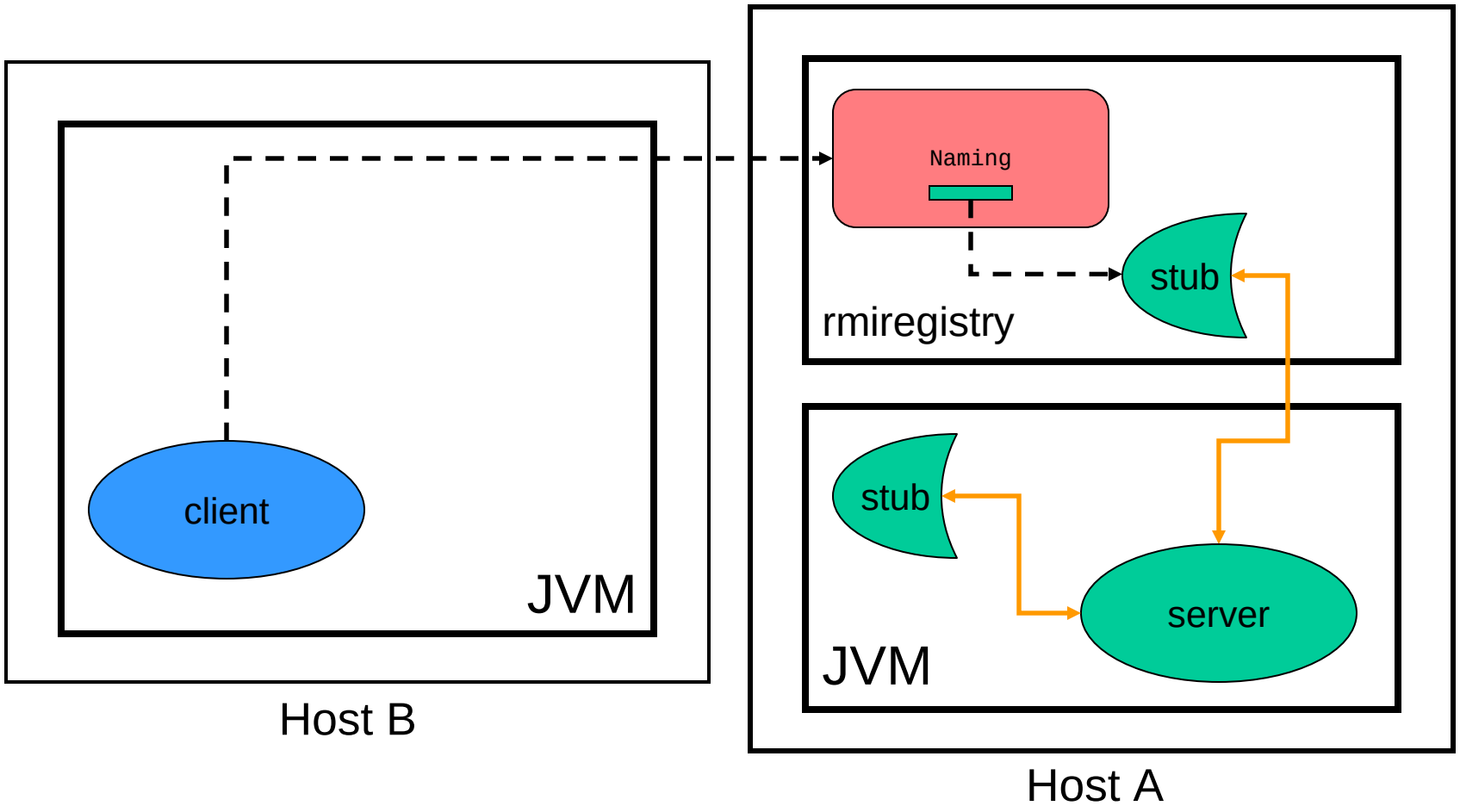
WriterClient

```
class WriterClient {
    public static void main(String[] args) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            String name = “//host:port/rwDictionary”;
            RWDictionaryServer d =
              (RWDictionaryServer) Naming.lookup(name);

            // Make a key,value pair and put it in the dictionary
            d.put(key, value);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

WriterClient

```
class WriterClient {
    public static void main(String[] args) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            String name = “//host:port/rwDictionary”;
            RWDictionaryServer d =
              (RWDictionaryServer) Naming.lookup(name);

            // Make a key,value pair and put it in the dictionary
            d.put(key, value);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

Naming.lookup

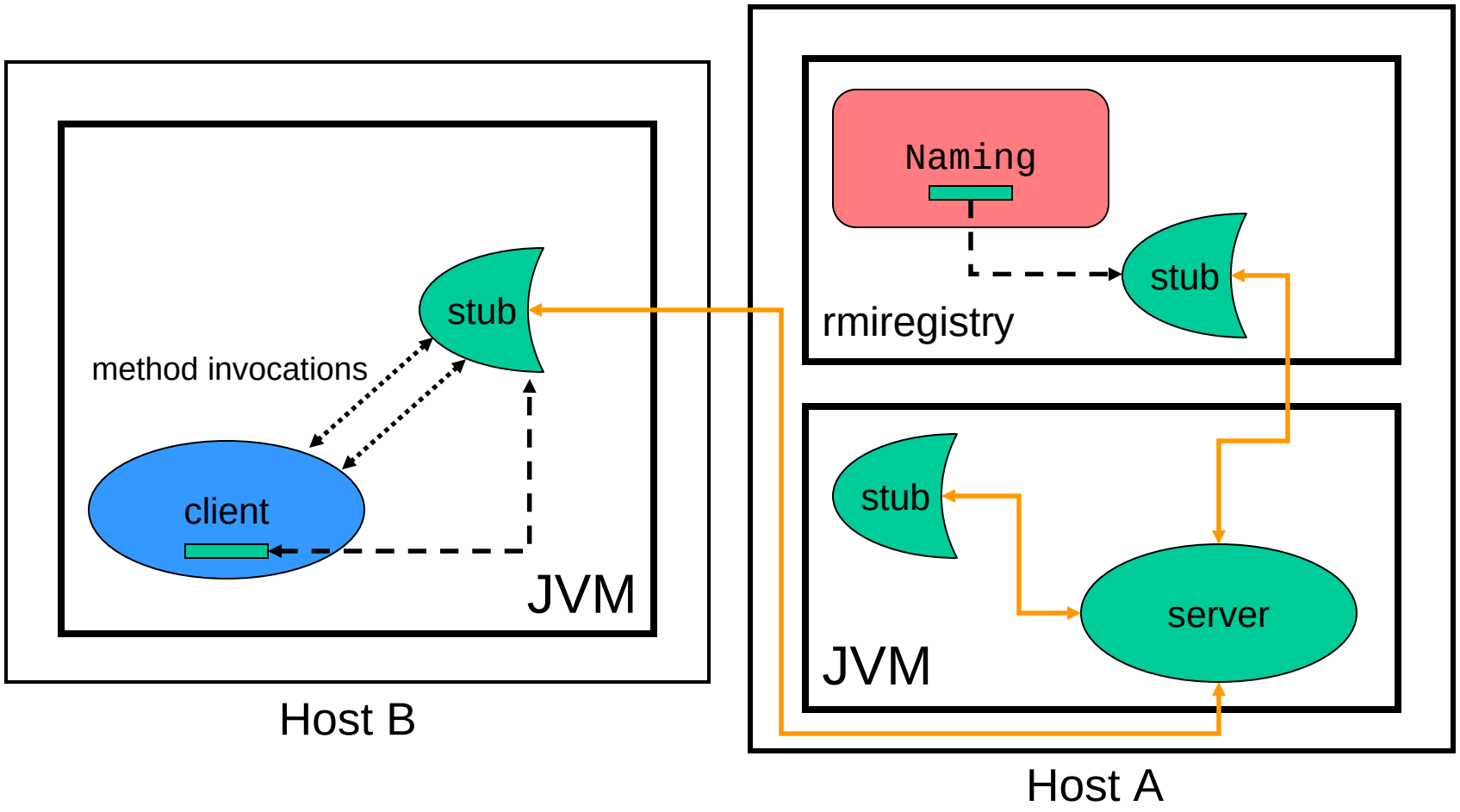


Remote method invocation on the stub

This returns a remote reference to the `RWDictionaryServerImpl` object—its stub:

- the `RWDictionaryServerImpl_Stub` is downloaded to the client's JVM from the registry's web server
- the stub knows the anonymous port on which the `RWDictionaryServer_Impl` is listening for method calls
- the `WriterClient` can then invoke methods on the stub, e.g., `put(String key, Data value)`

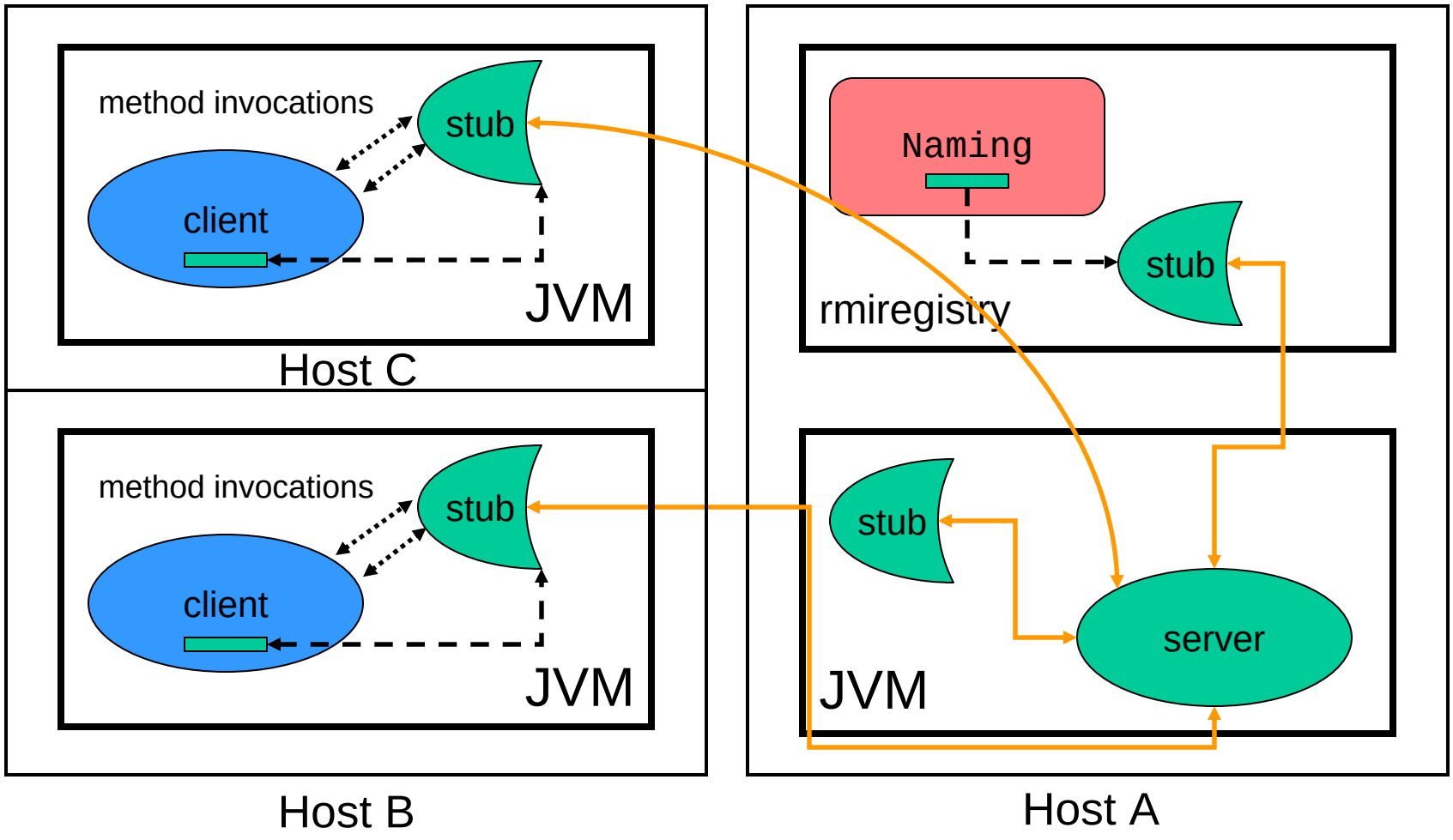
Distributed RWDictionary



ReaderClient

```
class ReaderClient {
    public static void main(String[] args) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            String name = “//host:port/rwDictionary”;
            RWDictionaryServer d =
              (RWDictionaryServer) Naming.lookup(name);

            // Check whether a key is in the dictionary
            Data value = d.get(key);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

Distributed RWDictionary



RMI and Threads

"A method dispatched by the RMI runtime to a remote object may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote method invocations to threads. Since remote method invocation on the same remote method may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe."

— Java RMI Specification, section 3.2

Summary

- in this part of the module, we have looked at the relationship between abstract concurrency ideas and their Java implementation:
  - Atomic Actions -- Java Memory Model
  - Mutual Exclusion -- `synchronized`
  - Condition Synchronisation -- `wait()`, `notify()`, `notifyAll()`
  - Distributed Processing -- `java.net` and `java.rmi`
- the Java implementation of concurrency is usually more complex (because of the need to handle things like exceptions); and
- the guarantees given by Java are often weaker than offered by the abstract model

The next lecture (after the break)

Proving Correctness

Suggested reading:

- Andrews (2000), chapter 2, sections 2.6–2.8;
- Ben-Ari (1982), chapter 3.