THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

Java RMI

SOFTENG 325 – Software Architecture

Andrew Meads

Last time, in SE325…

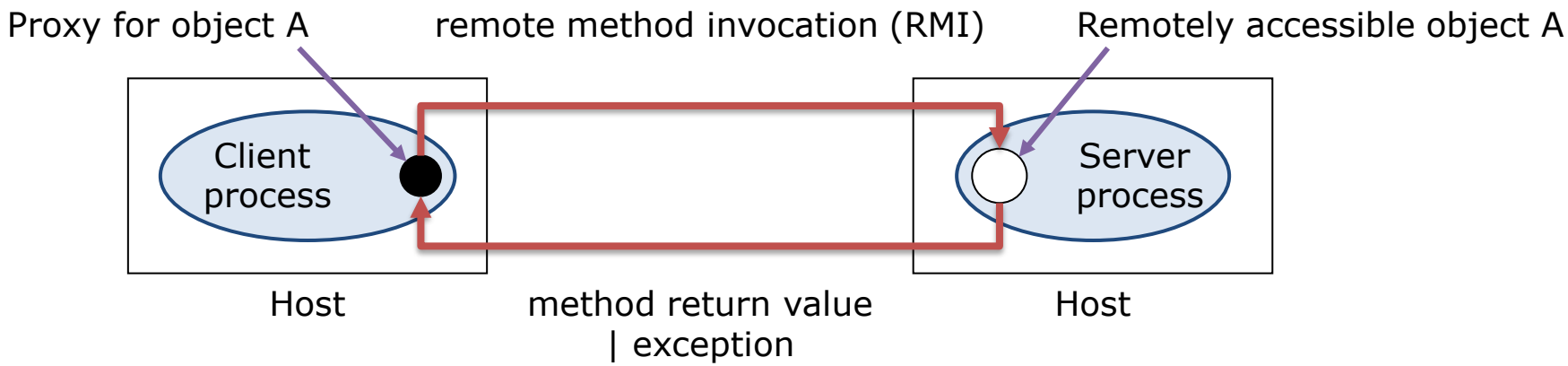
- Learned about what's to come in the course
- Learned about course assessments
- Introduction to distributed systems
  - TCP networking in Java
  - Serialization

Agenda

- Java RMI (Remote Method Invocation)
  - Distributed objects
  - Middleware
  - RMI middleware

Distributed objects

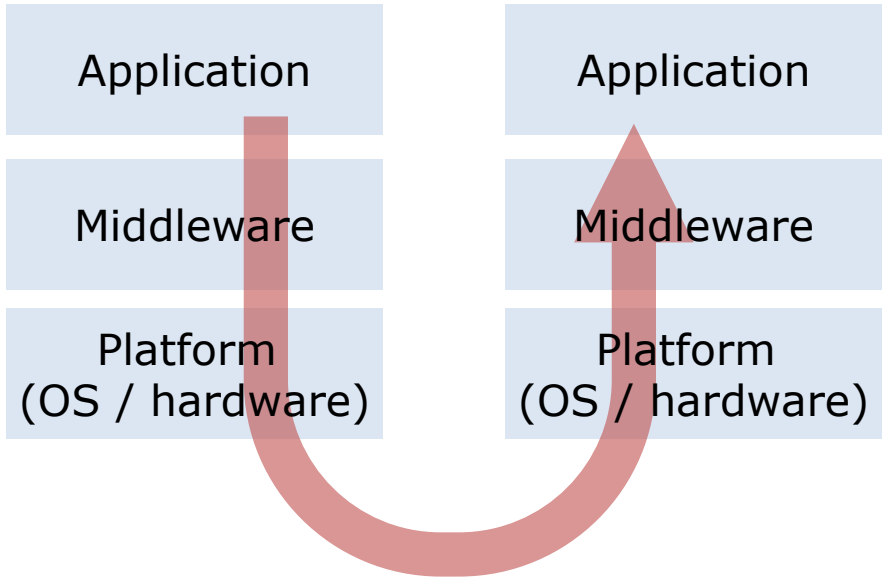
- Distributed object applications extend the familiar object-oriented programming model to the network
- Remote objects:
  - Can be invoked by Java programs on different machines
  - Are represented by local proxy objects that implement the same interfaces as the remote objects they represent



- What are some of the challenges in developing distributed object applications? How might invoking a method on a local object differ to a remote invocation?

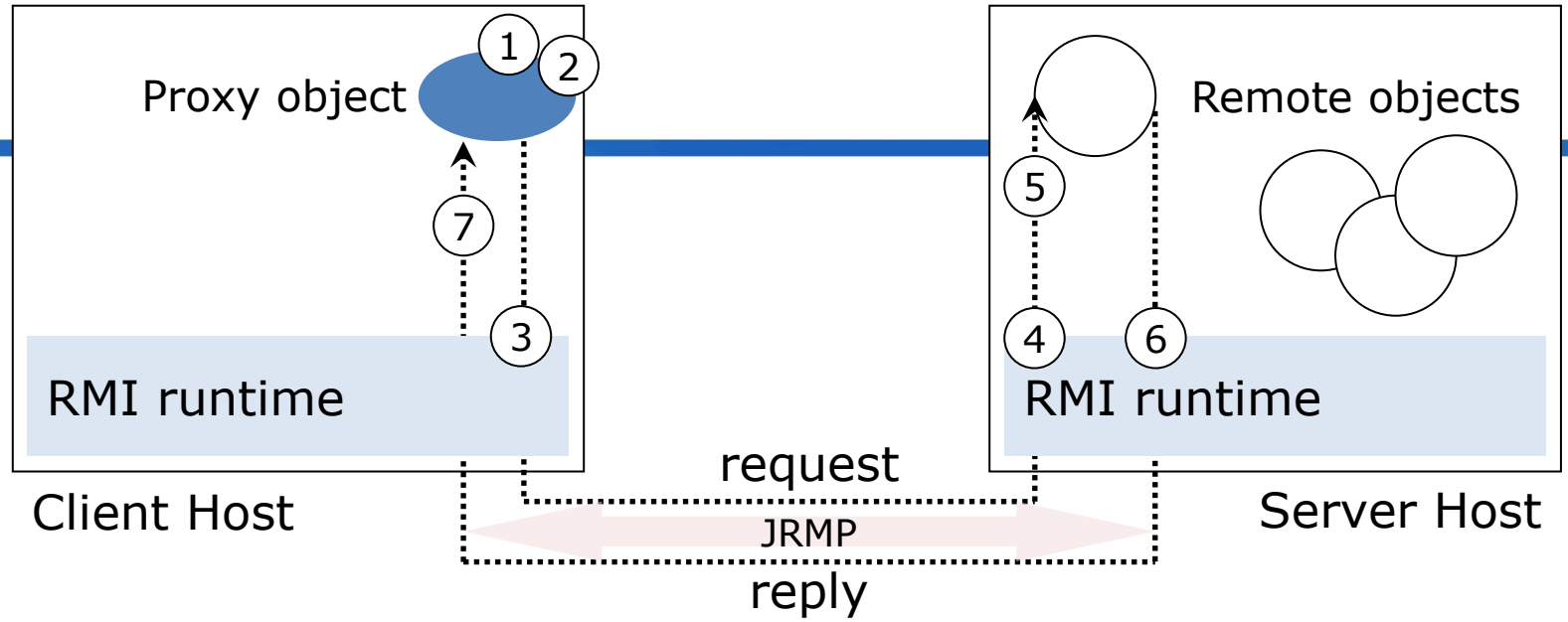
Middleware

- Middleware facilitates and manages interaction between applications, and provides:
  - A programming abstraction that hides some of the complexity associated with distribution
  - Infrastructure to implement the programming abstraction
  - A means to mask heterogeneity
- Key benefits of middleware include reuse, interoperability and portability



Java RMI (Remote Method Invocation)

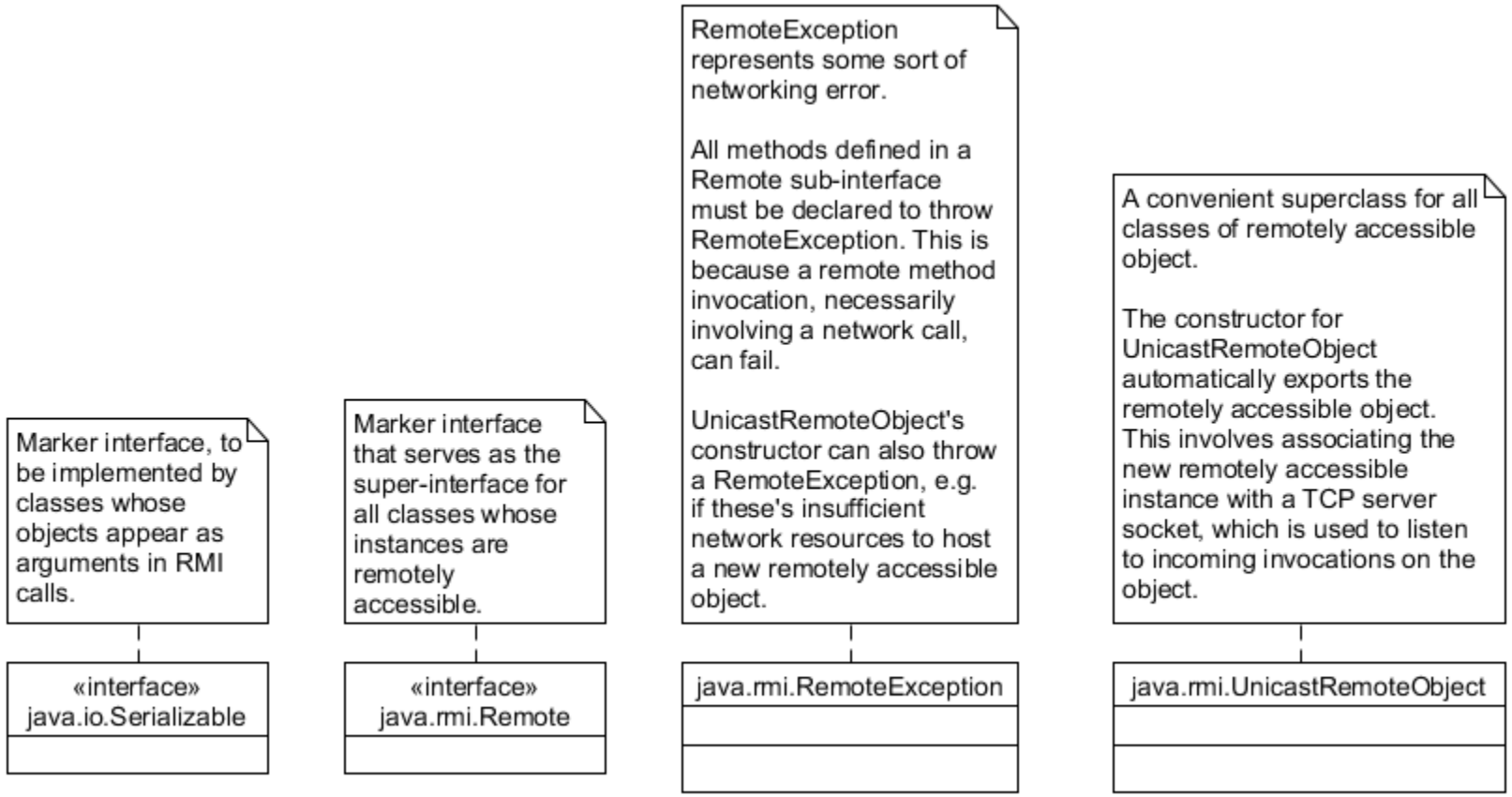
- Java RMI is an example of distributed object middleware; it provides the familiar object-oriented programming abstraction
- Java RMI masks distribution concerns by:
  - Generating proxy classes on demand, at run-time
  - Providing a Naming service that allows proxy instances to be registered and retrieved
  - Implementing a request-reply protocol layered on top of TCP
  - Providing resource management covering sockets, threads, and memory



1. Client application invokes method on proxy object
2. Proxy serializes method arguments
3. Proxy passes request to client-side run-time, which creates a JRMP message identifying the remote object and describing the method to invoke. Client-side run-time then acquires a TCP socket connected to the server and sends the request message.
4. Server-side run-time locates the object whose method is to be invoked.
5. The server-side runtime deserializes any arguments and executes the method
6. Method return value (or exception) is serialized and sent to the client
7. Return value / exception is deserialized; proxy method returns control to client thread with the result

A proxy object contains sufficient information to identify a remote object. Minimally, this includes the IP address of the server host, the port number of the server process in which the remote object resides, and an ID to identify the remote object within the server process.

Java RMI key entities



A "Hello World" example

- Let's build an RMI application allowing clients to get customized greetings from the server.
- First step: Define the *interface* for the remote object(s)

Our interface extends Remote – a marker interface indicating that instances of implementing classes can be exposed as remote objects.

```
public interface GreetingService extends Remote {

    String getGreeting(String name) throws RemoteException;

}
```

Argument and return types must either:
1. Be primitive; or
2. Be `Serializable`; or
3. Extend `Remote`

All methods must be declared to throw RemoteException.

A "Hello World" example

- Next, we can implement the interface to create our remote object class itself

Extending `UnicastRemoteObject` makes remote objects easier to implement – takes care of a lot of boilerplate code for us.

```
public class GreetingServiceServant extends UnicastRemoteObject implements GreetingService {

    public GreetingServiceServant() throws RemoteException {
    }

    @Override
    public String getGreeting(String name) throws RemoteException {
        return "Hello, " + name + "!";
    }
}
```

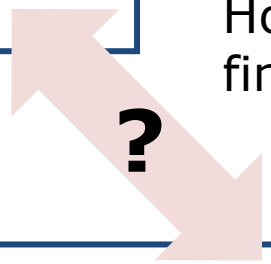
All constructors – even the default constructor – must throw `RemoteException`

A "Hello World" example

- And now, our client and server…

HelloWorldClient

```
try {
    GreetingService service = ???;

    String name = Keyboard.prompt("What is your name?");

    System.out.println(service.getGreeting(name));
} catch (RemoteException e) {
    e.printStackTrace();
}
```



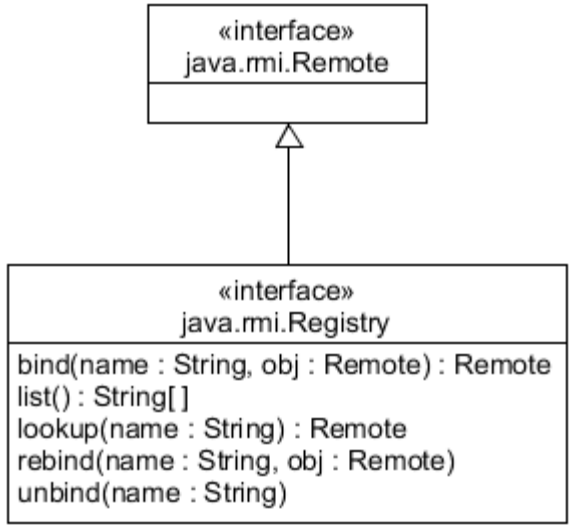
How does the client find the remote object?

HelloWorldServer

```
try {
    GreetingService greetingService = new GreetingServiceServant();
    System.out.println("Server up and running!");
} catch (RemoteException e) {
    e.printStackTrace();
}
```

Object discovery

- Somehow, clients need to obtain proxy objects so that they can invoke methods of remote objects
- Distributed object middleware typically offers a Naming service to store proxies
  - Servers register proxy objects with the Naming service
  - Clients lookup and retrieve proxy objects
- Java RMI provides a simple white-pages style naming service called the Registry



Object discovery

HelloWorldServer

```
try {
    GreetingService greetingService = new GreetingServiceServant();

    Registry lookupService = LocateRegistry.createRegistry(8080);
    lookupService.rebind("greetingService", greetingService);

    System.out.println("Server up and running!");
} catch (RemoteException e) {
    e.printStackTrace();
}
```

Creates a lookup service listening on the given port, and registers the remote object with it.

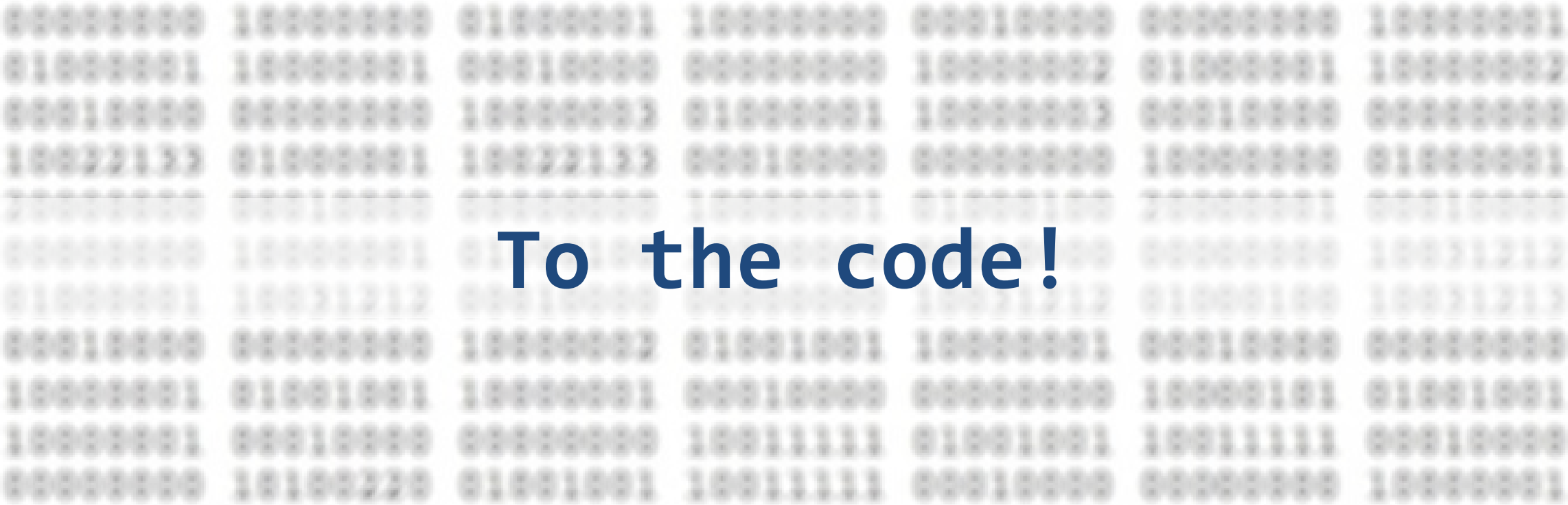
Object discovery

HelloWorldClient

```
try {
    Registry lookupService = LocateRegistry.getRegistry("localhost", 8080);
    GreetingService service = (GreetingService) lookupService.lookup("greetingService");

    String name = Keyboard.prompt("What is your name?");

    System.out.println(service.getGreeting(name));
} catch (RemoteException e) {
    e.printStackTrace();
} catch (NotBoundException e) {
    e.printStackTrace();
}
```

Thrown if an object with the given name is not found.

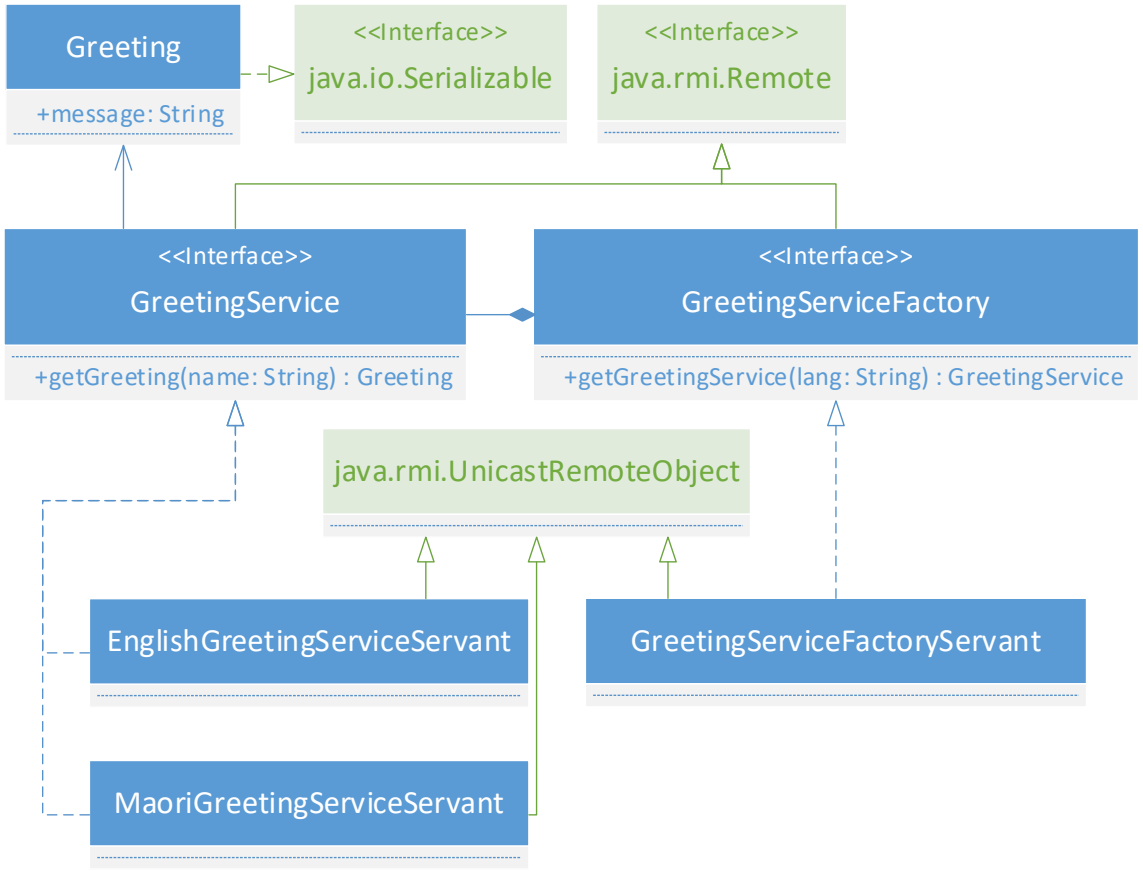
Gets the lookup service listening on the given host / port, and finds the remote object with the given name.

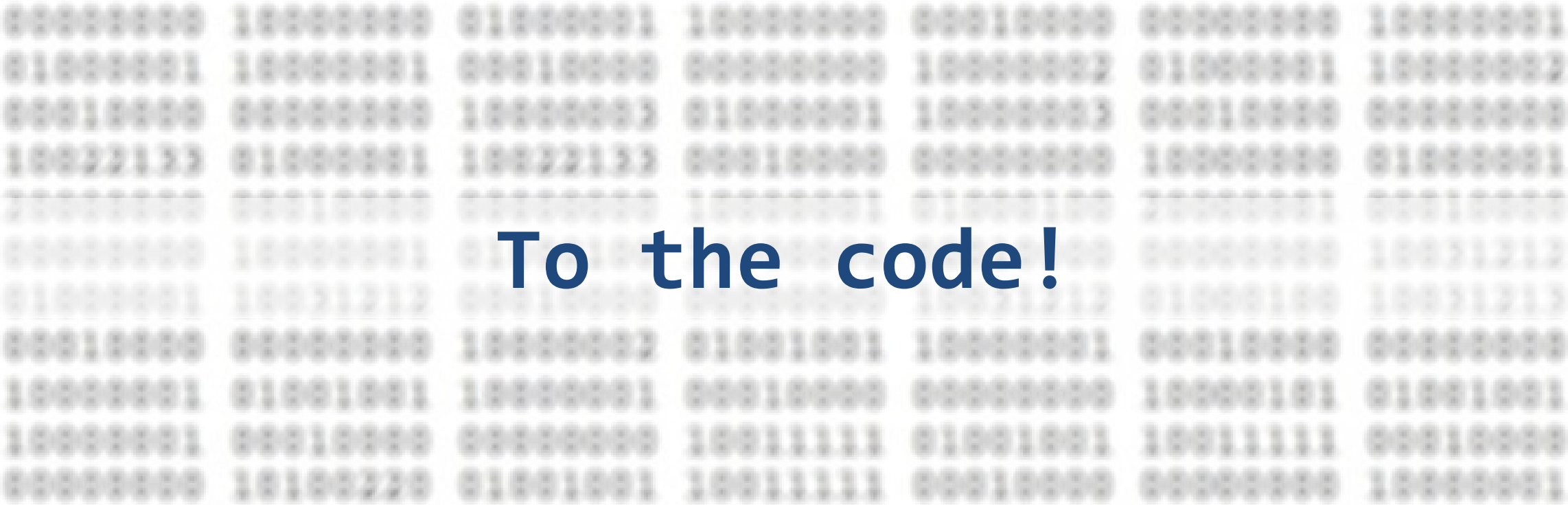


To the code!

Slightly more complex example

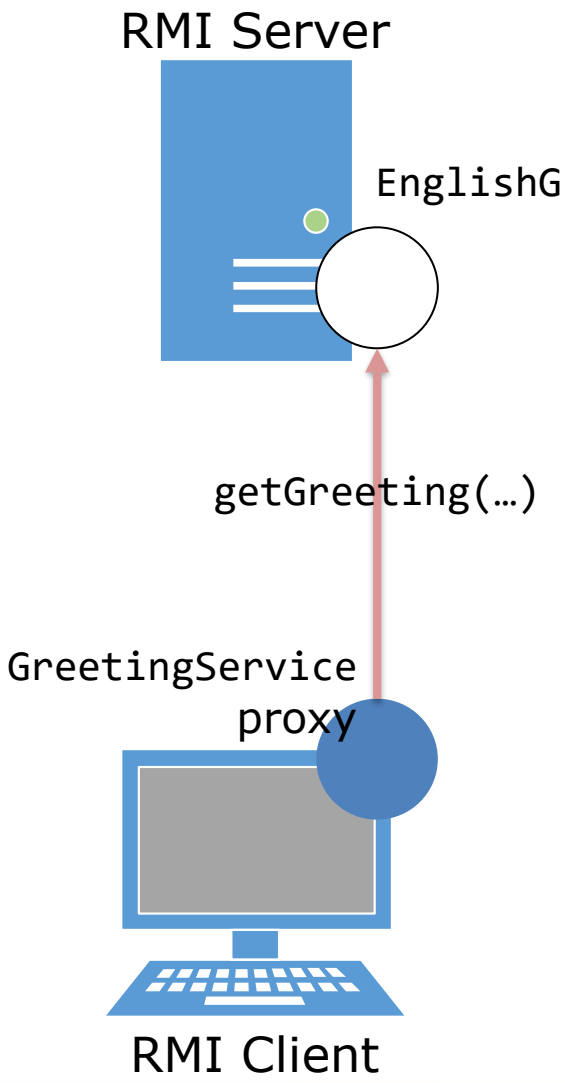
- Let's extend our example to include multilingual support
  - Each language will be handled by a different remote object
  - We will have a remote `GreetingServiceFactory` object to return the correct `GreetingService` based on the language choice





To the code!

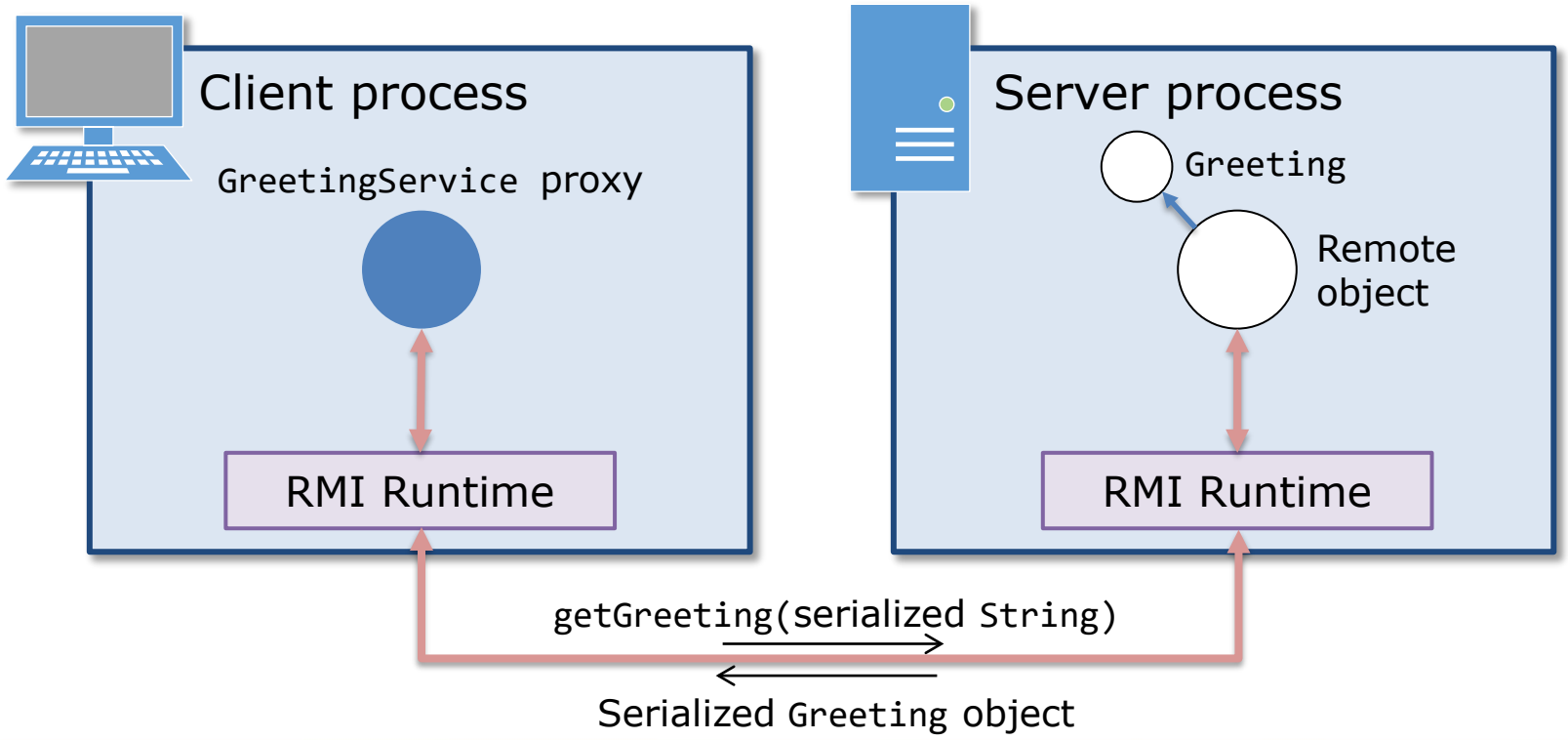
Parameter passing – Serializable data




```
public class EnglishGreetingServiceServant
        extends UnicastRemoteObject implements GreetingService {


    …

    @Override
    public Greeting getGreeting(String name) throws RemoteException {
        return new Greeting("Hello, " + name + "!");
    }
}
```

Parameter passing – Serializable data

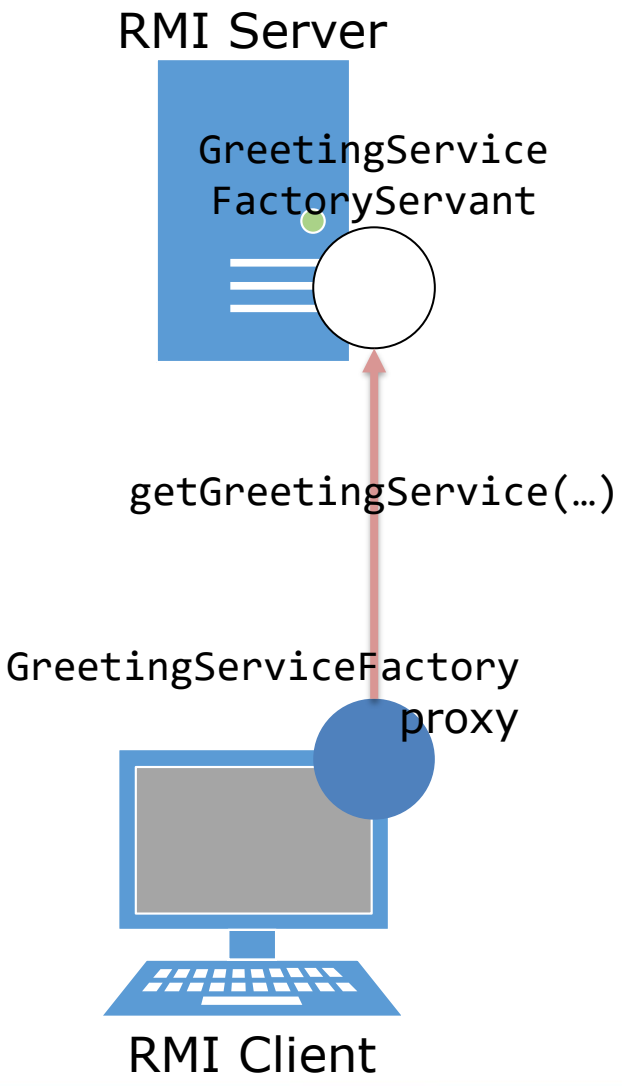
```
GreetingService greetingServiceProxy = …;
String name = …;

Greeting greeting = greetingServiceProxy.getGreeting(name);
greeting.setMessage("...");
```



- The serialization mechanism:
  - Serializes the `Greeting` object in the server
  - Sends the object in its serialized form to the client
  - Deserializes the `Greeting` in the client JVM, creating a local copy

What effect will modifying the greeting have on the server?

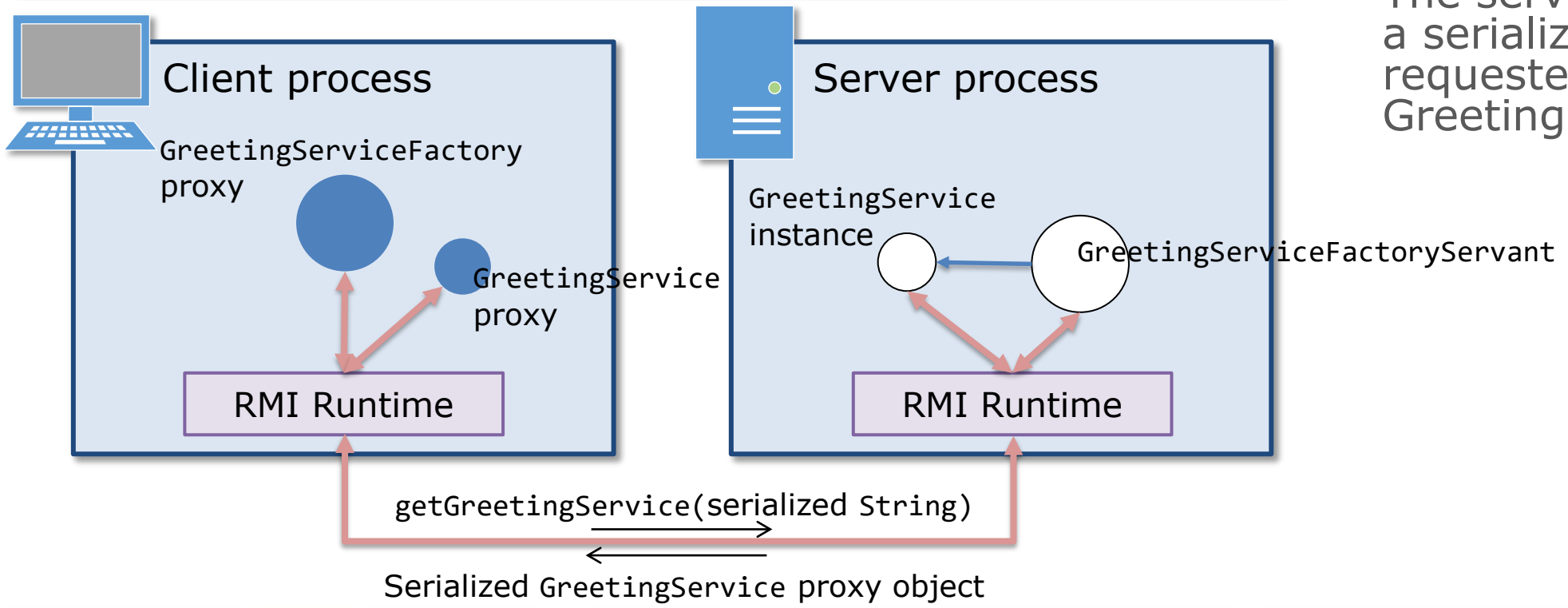
Parameter passing – Remote objects




```
public class GreetingServiceFactoryServant
        extends UnicastRemoteObject implements GreetingServiceFactory {

    private Map<String, GreetingService> services = new HashMap<>();

    public GreetingServiceFactoryServant() throws RemoteException {
        services.put("english", new EnglishGreetingServiceServant());
        services.put("maori", new MaoriGreetingServiceServant());
    }

    @Override
    public GreetingService getGreetingService(String language)
        throws RemoteException {
        return services.get(language);
    }
}
```

Parameter passing – Remote objects

```
GreetingServiceFactory greetingFactoryProxy …;
String language = …;
GreetingService greetingServiceProxy =
        greetingFactoryProxy.getGreetingService(language);
String name = …;
Greeting greeting = greetingServiceProxy.getGreeting(name);
```

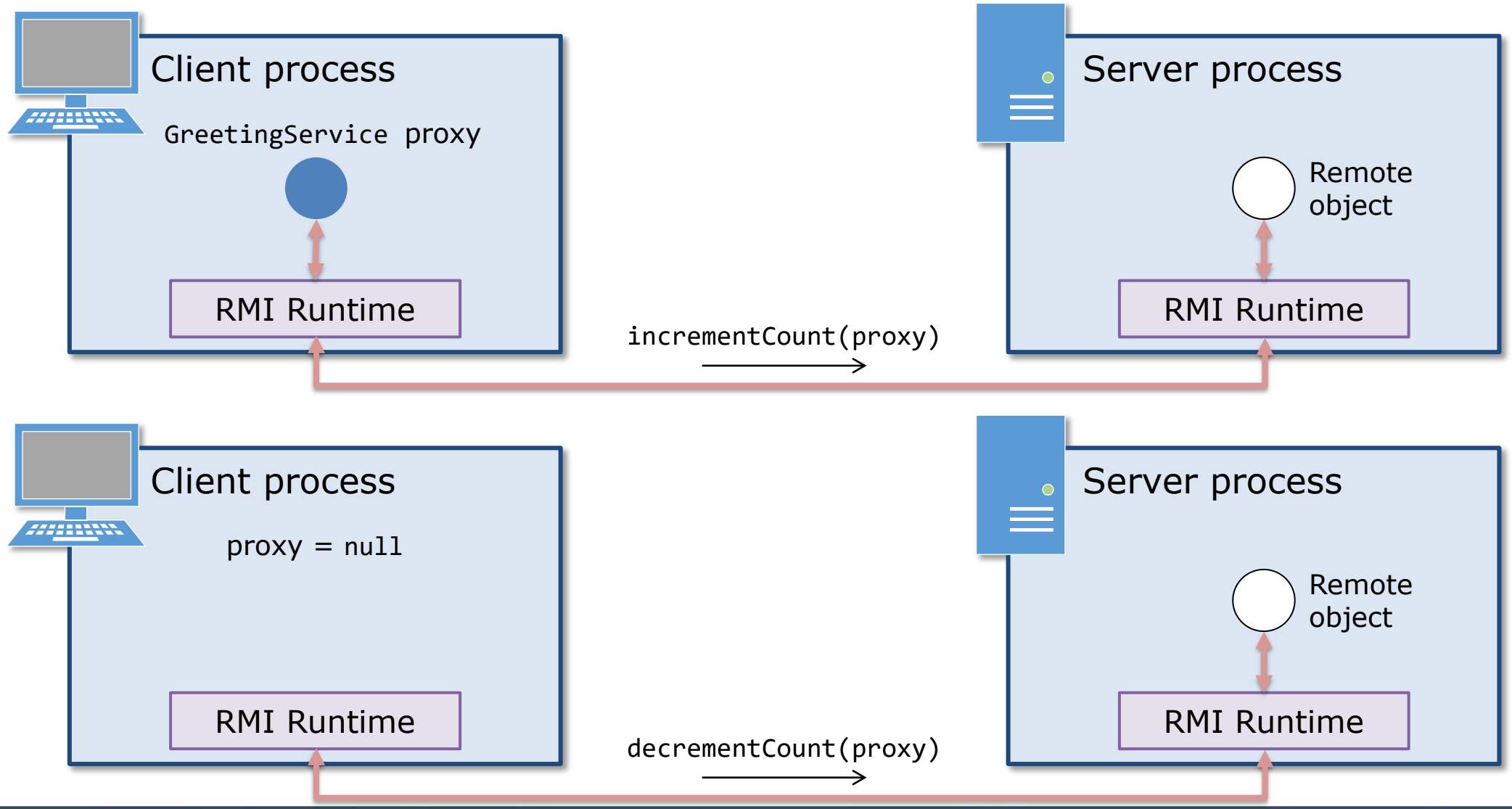


- The serialization mechanism serializes the String parameter on the client and sends it to the server
- The server then returns a serialized *proxy* to the requested remote GreetingSrervice object.

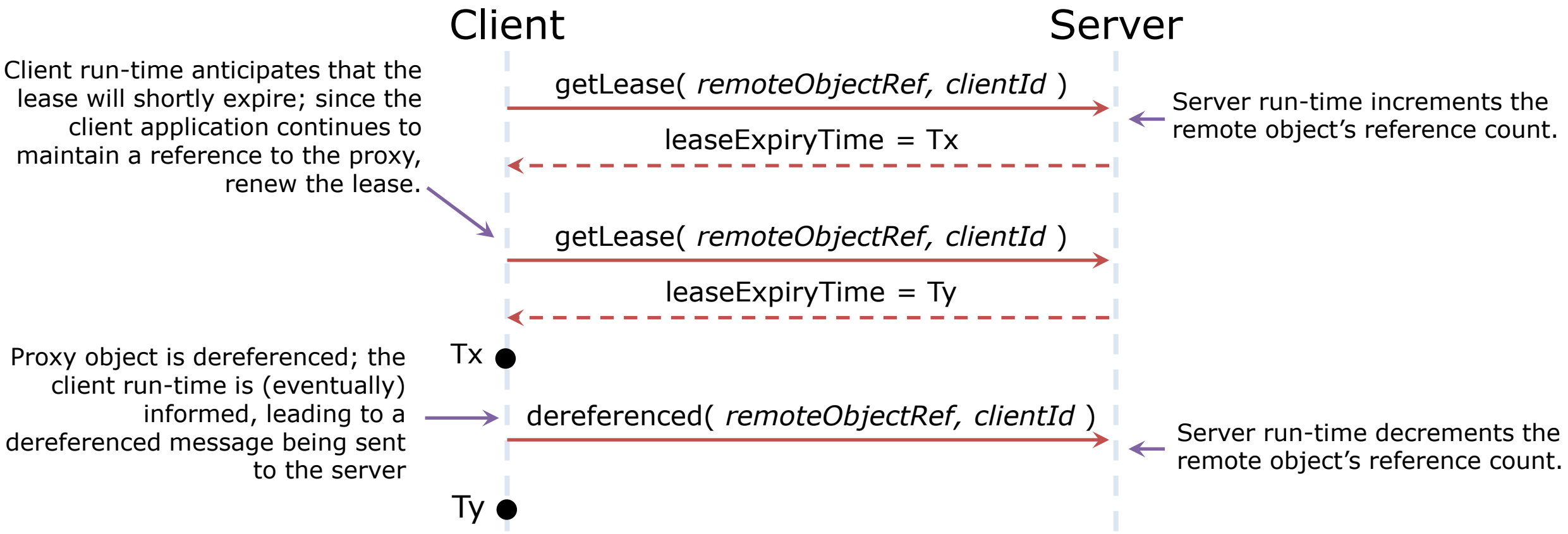
Resource management

- The RMI middleware is responsible for managing resources efficiently; resources include
  - Memory
    - Just like memory is reclaimed when a local object is no longer referenced, memory used by unreachable remotely accessible objects should be reclaimed
    - There is a need for distributed garbage collection
  - Sockets and network resources
    - Sockets and network connections are expensive to establish and maintain; the middleware should carefully manage sockets
  - Threads

Distributed garbage collection – a solution



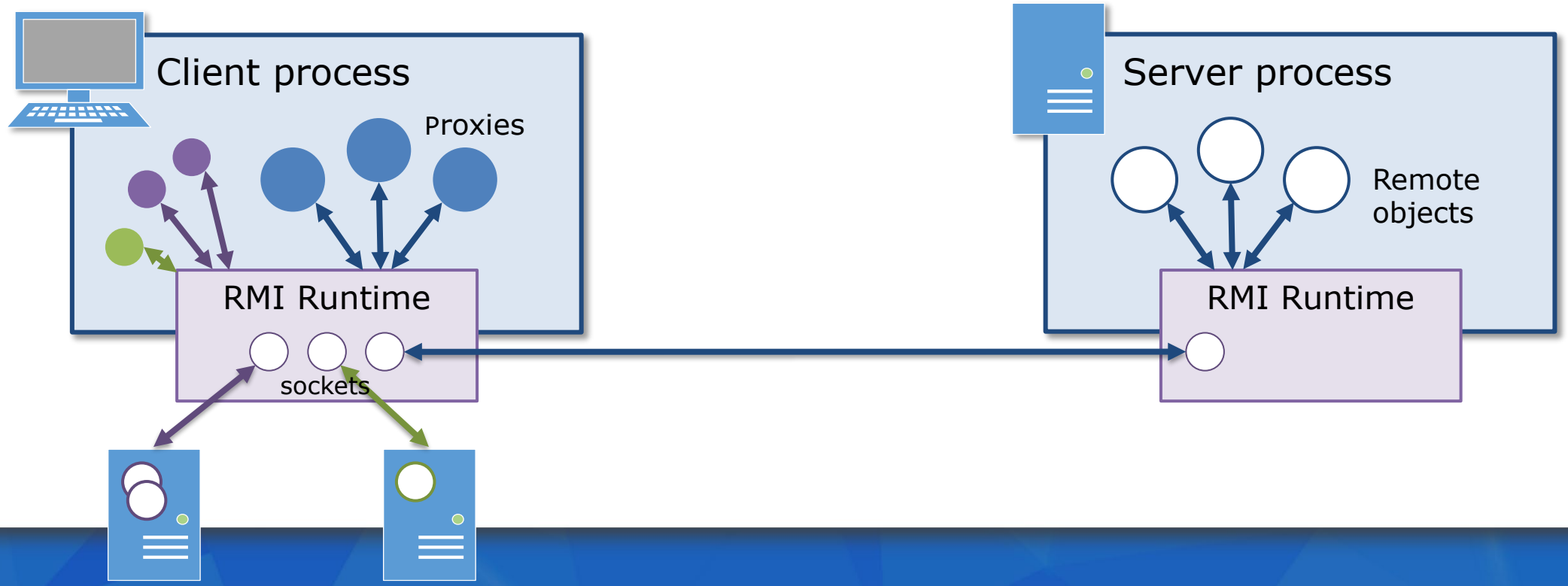
An alternative – leasing



This solution helps to prevent resource starvation on servers. Note too that the dereferenced message need not arrive at the server – in this case, the server would decrement the remote object's reference count at time Ty.

Socket sharing

- It would be grossly inefficient for each proxy object to maintain a dedicated socket connection
- Instead, when a proxy's method is called, the middleware uses an existing socket connection to the server (if one exists and isn't in use); otherwise a new connection is established



Invocation semantics

- With a method call on a local object, the method is executed exactly once
- RMI middleware cannot guarantee exactly-once semantics, and typically offers at-most-once semantics
  - Where a RMI call returns successfully (i.e. it doesn't throw a `RemoteException`), the method was executed once
  - In the event of a `RemoteException`, the remote method may or may not have been executed

What have we learned today?

- Middleware
  - A layer that has important benefits: reuse, interoperability and portability
  - Simplifies application development as developers can focus on application logic and not networking complexity
  - Helps manage resources efficiently
- RMI middleware
  - Aims to make remote method invocation as simple as invoking a method on a local object
  - Provides object-invocation programming abstraction but cannot completely mask a distributed environment
  - Parameter passing and invocation semantics are different for remote invocations, object discovery is required in a distributed environment, and Remote interfaces are necessitated for remotely accessible objects

Lab 01

- Review Maven
- Critique a Java TCP client/server application
- Review a shared whiteboard application which uses Java RMI
- Develop a Java RMI application for maintaining concerts
- Reflect on the Java RMI application, including threading issues

Next week

- Lab 01
- HTTP & Java Servlets
- Web services – SOAP vs REST
- REST in Java using JAX-RS