JAVA™ 2

SUN™ CERTIFIED PROGRAMMER & DEVELOPER

15

Networking Issues

CERTIFICATION OBJECTIVE

- Understand Networking Issues

CERTIFICATION OBJECTIVE

Understand Networking Issues

It is better to know some of the questions than all of the answers.

–James Thurber

Good questions outrank easy answers.

–Paul A Samuelson

If you don't ask the right questions, you don't get the right answers. A question asked in the right way often points to its own answer. Asking questions is the ABC of diagnosis. Only the inquiring mind solves problems.

–Edward Hodnett

Clever as you are, I bet you've figured out where this is heading…the Developer exam is about *you* figuring out solutions to the problem/specification you're given as your assignment. So, any attempt on our part to offer suggested potential solutions would, in our humble opinion, be defeating the whole point of the certification. However, given that this *is* a book about preparing for the exam, we can offer you questions. Things to think about. But we will start with a briefing on the core technologies involved: Serialization, Sockets, and RMI. There's far more to learn about these than we could possibly say here, so we're not even going to *attempt* to give you a crash-course. We're assuming that you're familiar with the technologies, and that you'll do whatever research and experimentation you need to learn to use them correctly. We will, however, do a simple review and then look at issues you'll need to consider when you build your project.

RMI and Sockets

As of this writing, the Developer exam expects you to know about networking. Well, not just *know* but actually *develop* a network server that allows remote clients to get information from a database (which you will *also* write).

Normally, building a simple network server presents you with two choices: RMI or Sockets. If your assignment asks you to make a choice, rest assured that there is *not one right answer*. You will need to think through the tradeoffs, make a choice, and document your decision.

One simple way to look at the difference is this:

Sockets are low-level, RMI is high-level.

In other words, RMI is a higher-level system that *uses* Sockets underneath. Whichever you choose, you'll need to be very comfortable with it, and you'll need to justify your choice.

Serialization

Somehow you're going to have to move a client *request*—made on one machine—across a wire to another machine. For your assignment, that "machine" might be only a virtual machine running on the same physical computer, but the Big Issues are the same whether the two machines (the client and the server) are on the same physical box or not. Two JVM's might as *well* be on two different boxes, with one key exception—the classpath. If two instances of a JVM are started on the same computer, they may well have access to the *same stuff*, and sometimes that masks a problem with your application. So, whatever you do, test test test on two different *physical* machines if you can.

What form is the client request? Well, remember from Chapter 11 when we looked at the Horse Cruise system. A client might want to request a cruise based on a certain date or horse criteria (easy horse, short horse, fast horse, etc.), or perhaps both. Ultimately, that request can take any form before you ship it from the client to the server; that's up to you, but let's say you're going to use a String. That String needs to be packaged up, shipped out, and land at the other end. When it's picked up at the other end, *the other end has to know how to use it.*

So we're really looking at *two* issues: how to pack and unpack it for shipping, and then how to make sure it makes sense to the program on the other end (the server). The packing and unpacking is easy—Serialization. Whatever object(s) you ship over the wire, they can be sent simply as *flattened* objects (*serialized*) and then they get brought back to life (*deserialized*) on the other end, and once again become real objects on the heap. So the object traveled from one heap to another. Well, it wasn't even the *object* that traveled, but a *copy* of the object. (We covered serialization in Chapter 6.)

OK, so the client makes a request for, say, a Horse Cruise on such and such a date. Now what? We put the client request into a String, serialize it, and ship it out (we haven't yet said whether this would be through RMI or straight Sockets) and the server picks it up, deserializes it, and uses it as an object. *Now* what? The client obviously needs a *result* from the request. Whatever that result actually is, you'll stuff

it in an object (or group of objects) and ship it back following the same process—serialize it, ship it to the client, client deserializes it and uses it in some meaningful way (most likely, presenting the Horse Cruise search results in a GUI).

So now we know what form the request and result take (serialized objects), but we still need to know *how to ship it from point A to point B* (in other words, from client to server and from server to client). That leaves us with only one real question: do we use Sockets or RMI?

Sockets

Given that Sockets are simply the end-points of a connection between two devices, you aren't limited to shipping only serialized objects over the wire. In fact, the Socket has no idea *what's* coming—it just sees a stream of bytes. When the server gets the bytes, it's up to your server code to figure out what those bytes are supposed to mean. Are they in fact serialized objects? Then deserialize them…but in what order are they coming over? The server needs to know. And if they're *not* serialized objects, the server needs to know exactly what *is* in those bytes, in exactly which order, so it can do the appropriate *read* and get the bytes into some form a little more useful (Strings, numbers, etc.). And for that, you'll need a *protocol*. The client code developer and the server code developer will have to get together in advance and come to an agreement on what these bytes mean, in other words, how they're supposed to be interpreted. Of course, in the Developer exam, you're writing both the client and the server, so you only have to agree with *yourself*.

RMI

The beauty of RMI is that the protocol is already agreed on by both ends—it's just objects moving from one place to another (*copied* into the new location, but we're just going to say *moved* because it's easier to think about). In other words, we've already decided on the protocol for what the bytes mean, and the protocol is serialized objects. And since we've established *that,* then the client and server don't even have to do the serialization/deserialization—RMI takes care of it.

The tradeoffs, then, are already shaping up: using Sockets lets you have whatever protocol you want, but you have to do all the heavy lifting, while using RMI restricts the protocol to serialization. But with that flexibility removed, RMI can do most of the heavy lifting. By *heavy lifting*, we mean things like establishing the Socket connections, packing up the bytes, sending an output stream from one place to another, then receiving it and unpacking it and so on.

RMI is much simpler than Sockets. But simplicity never comes for free, of course, so it's also a little slower. You have to decide in all of this which is most important, but here's a hint: think Team Work. Again, there's no right answer; the assessor isn't going to prefer one over the other, because it depends on your goal and need (or in some cases, all things being equal, just which you prefer). But whichever you choose, you need to justify in your documentation (and possibly on the essay portion of the exam) why you chose one over the other.

The rest of this document looks at some of the things you'll need to think about when implementing your solution. They are in no particular order, so don't infer any relative importance from the order in which they're listed.

Questions to Ask Yourself

Ask yourself these questions as you design and implement your Exam solution.

- Serialization sends the entire object graph from point A to point B, for all instance variables that are not marked transient. Is your object large? Do you really *need* all that data to be shipped?
- Have you marked everything transient except the state you truly need on the other end?
- Have you investigated leaving most of the state transient and then reconstructing it by stepping into the deserialization process? The process of implementing your own private `readObject()` method can help. (Think of a private `readObject()` as kind of like a constructor, except instead of constructing an object for the first time, the private `readObject()` is involved in *re*constructing the object following serialization.)
- Are the superclasses of your serialized class also serializable? If not, they'll need a no-argument constructor, so be certain that this fits your design.
- Do you have any final transient variables? Be careful! If you use *blank* finals, you're in big trouble because the variable will come back to life *with a default value* since the constructor won't run when deserializing an object. And since it's final, you'll be stuck with that value.
- Have you thought about versioning issues? When an object is serialized, remember, the class needs to be present at the location where the object is being *de*serialized. What happens if the class has been modified? Will the

object still be usable if it was serialized from one version of the class and deserialized using a *different* version? Consider declaring an explicit serial version ID in all your serializable classes to reduce versioning issues.

- What happens if your database schema changes? Will your protocol have to change as well? Will your remote interface need to change?
- Have you looked for relevant design patterns? Not that we're suggesting anything, but have you looked into, oh, I don't know, say, the Command and Proxy patterns?
- Have you thought about *how* the client should get the stub class? When the client does a lookup in the RMI registry, remember, it's the stub object that gets shipped, and the client will need it. Dynamic code downloading is by far the coolest and most flexible, but unless your project specification appears to need it, it may well be more work than you need to provide.
- Have you thought about how the classes for your return types and arguments will get shipped? Are you certain that both the client and server will have all the classes they need for objects that will be shipped across from requests and results? Remember, if you've followed the rules for maintainability, you're most likely using interface types in your code, even though at runtime you'll be deserializing objects from classes that implement the interface…so the class of the implementation object needs also to be on the machine that's deserializing the object.
- How should you start the RMI registry? You can start it from the command-line, or you can start it programmatically, but you'll need to decide what's right (unless your assignment instructions/specification explicitly requires one way over the other).
- How should you shut *down* the registry?
- What happens if you need a remote object to stop being available to a client, while everything is still running? Have you looked into different options for binding and unbinding objects in the registry?
- How does the server know when a client is finished with an object? Does it even *need* to?
- What happens if a client crashes *after it locked a record*. How will you know the client is gone versus just taking a long time? Will you use a timeout

mechanism like a distributed lease that the client has to renew periodically to say, "I'm still alive!"? What about the state of the record?

- Have you looked at the `java.rmi.server.Unreferenced` interface to see if it does anything you can use?
- Have you considered bandwidth limitations if a client request turns up a lot of results?
- If you're using Sockets, do you have any potential issues with platform-specific line endings?
- How will the server know that a particular client asking to update a record is in fact the *same* client that got the lock on that record? In other words, how will you identify the client to the server in a unique way? Don't count on using the thread ID; remember that RMI makes no guarantees that it will keep the same thread for each request from a particular client.
- For that unique client ID, can you use something like web browsers use like a cookie mechanism? If so, *how* can you guarantee that one client won't have the same ID# as another client? Have you considered how you might generate your own unique identifiers? Would `random()` alone do the trick?

exam Watch

There's a class you can look at in the Jini distribution, com.sun.jini.reggie.RegistrarImpl.java, that generates universally unique identifiers.

- Have you considered the possibility of distributed deadlock?
- How will you provide thread-safety in your server? Will it be at the remote object level or in your actual database? Remember, RMI doesn't guarantee that more than one thread won't be accessing your remote object.
- Have you thought about how much caffeine you'll need to complete this project?
- Have you begun to forget why you even wanted to *be* a Sun Certified Java Developer? (That's natural. We all go through that.)

For the tricky networking issues you'll encounter as you get into the specifics of your project, the best resource we can suggest is the Big Moose Saloon at javaranch.com. The saloon is a threaded discussion board with more than 50,000

posts in the Developer Certification section. Anything you might struggle with has already been struggled with by dozens of others who are willing to offer guidance.

Key Points Summary

We're not going to summarize the points we made under the Questions to Ask Yourself heading; they're *already* bullet points. But here's a quick summary of the key points around RMI and Sockets:

- Your exam assignment will require networking. Most likely you'll be asked to choose between RMI and Sockets.
- Sockets are low-level, RMI is high-level.
- RMI uses Sockets to do its work.
- To move an object from one JVM to another, you need to serialize it. (See Chapter 6.)
- Serialization flattens an object by packaging up the object's state.
- An object's serializable state consists of all nontransient instance variables.
- Sockets are the end-points of a connection between two networked devices.
- You can send a stream of bytes between Socket connections, but you'll need an agreed-upon protocol to know what those bytes mean.
- RMI uses serialization as the protocol, so you don't need to read and manipulate the bytes in your code.
- RMI is simpler to implement than using plain Sockets.
- Sockets offer more flexibility in protocol than does RMI.
- Ask yourself all the questions on the "Ask Yourself" list.