2.1 Lecture Summary

2.1 Introduction to Sockets

Lecture Summary: In this lecture, we learned about *client-server programming*, and how two distributed Java applications can communicate with each other using *sockets*. Since each application in this scenario runs on a distinct Java Virtual Machine (JVM) process, we used the terms "application", "JVM" and "process" interchangeably in the lecture. For JVM A and JVM B to communicate with each other, we assumed that JVM A plays the "client" role and JVM B the "server" role. To establish the connection, the main thread in JVM B first creates a ServerSocket (called socket, say) which is initialized with a designated URL and port number. It then waits for client processes to connect to this socket by invoking the socket.accept() method, which returns an object of type Socket (called s, say) . The s.getInputStream() and s.getOutputStream() methods can be invoked on this object to perform read and write operations via the socket, using the same APIs that you use for file I/O via streams.

Once JVM B has set up a server socket, JVM A can connect to it as a client by creating a Socket object with the appropriate parameters to identify JVM B's server port. As in the server case, the getInputStream() and getOutputStream() methods can be invoked on this object to perform read and write operations. With this setup, JVM A and JVM B can communicate with each other by using read and write operations, which get implemented as messages that flow across the network. Client-server communication occurs at a lower level and scale than MapReduce, which implicitly accomplishes communication among large numbers of processes. Hence, client-server programming is typically used for building distributed applications with small numbers of processes.

Optional Reading:

1. Java tutorial titled Lesson: All About Sockets

2.2 Lecture Summary

2.2 Serialization and Deserialization

Lecture Summary: This lecture reviewed *serialization* and *deserialization*, which are essential concepts for all forms of data transfers in Java applications, including file I/O and communication in distributed applications. When communications are performed using input and output streams, as discussed in the previous lecture, the unit of data transfer is a sequence of bytes. Thus, it becomes important to *serialize* objects into bytes in the sender process, and to *deserialize* bytes into objects in the receiver process.

One approach to do this is via a *custom* approach, in which the programmer provides custom code to perform the serialization and deserialization. However, writing custom serializers and deserializers can become complicated when nested objects are involved, e.g., if object x contains a field, f3, which points to object y. In this case, the serialization of object x by default also needs to include a serialization of object y. Another approach is to use XML, since XML was designed to serve as a data interchange standard. There are many application frameworks that support conversion of application objects into XML objects, which is convenient because typical XML implementations in Java include built-in serializers and deserializers. However, the downside is that there can be a lot of metadata created when converting Java objects into XML, and that metadata can add to the size of the serialized data being communicated.

As a result, the default approach is to use a capability that has been present since the early days of Java, namely Java Serialization and Deserialization. This works by identifying classes that implement the Serializable interface, and relying on a guarantee that such classes will have built-in serializers and deserializers, analogous to classes with built-in toString() methods. In this situation, if object x is an instance of a serializable class and its field f3 points to object y, then object y must also be an instance of a serializable class (otherwise an exception will be thrown when attempting to serialize object x). An important benefit of this approach is that only one copy of each object is included in the serialization, even if there may be multiple references to the object, e.g., if fields f2 and f3 both point to object y. Another benefit is that cycles in object references are handled intelligently, without getting into an infinite loop when following object references. Yet another important benefit is that this approach allows identification of fields that should be skipped during the serialization/deserialization steps because it may be unnecessary and inefficient to include them in the communication. Such fields are identified by declaring them as *transient*.

Finally, another approach that has been advocated for decades in the context of distributed object systems is to use an *Interface Definition Language (IDL)* to enable serialization and communication of objects across distributed processes. A recent example of using the IDL approach can be found in Google's Protocol Buffers framework. A notable benefit of this approach relative to Java serialization is that protocol buffers can support communication of objects across processes implemented in different languages, e.g., Java, C++, Python. The downside is that extra effort is required to enable the serialization (e.g., creating a .proto file as an IDL, and including an extra compile step for the IDL in the build process), which is not required when using Java serialization for communication among Java processes.

Optional Reading:

1. Wikipedia article on Serialization

2.3 Lecture Summary

2.3 Remote Method Invocation

Lecture Summary: This lecture reviewed the concept of *Remote Method Invocation (RMI)*, which extends the notion of method invocation in a sequential program to a distributed programming setting. As an example, let us consider a scenario in which a thread running on JVM A wants to invoke a method, foo(), on object x located on JVM B. This can be accomplished using sockets and messages, but that approach would entail writing a lot of extra code for encoding and decoding the method call, its arguments, and its return value. In contrast, Java RMI provides a very convenient way to directly address this use case.

To enable RMI, we run an RMI client on JVM A and an RMI server on JVM B. Further, JVM A is set up to contain a stub *object* or *proxy object* for remote object x located on JVM B. (In early Java RMI implementations, a *skeleton object* would also need to be allocated on the server side, JVM B, as a proxy for the shared object, but this is no longer necessary in modern implementations.) When a stub method is invoked, it transparently initiates a connection with the remote JVM containing the remote object, x, serializes and communicates the method parameters to the remote JVM, receives the result of the method invocation, and deserializes the result into object y (say) which is then passed on to the caller of method x.foo() as the result of the RMI call.

Thus, RMI takes care of a number of tedious details related to remote communication. However, this convenience comes with a few setup requirements as well. First, objects x and y must be *serializable*, because their values need to be communicated between JVMs A and B. Second, object x must be included in the *RMI registry*, so that it can be accessed through a global name rather than a local object reference. The registry in turn assists in mapping from global names to references to local stub objects. In summary, a key advantage of RMI is that, once this setup in place, method invocations across distributed processes can be implemented almost as simply as standard method invocations.

Optional Reading:

1. Tutorial on Java RMI

2. Wikipedia article on Java remote method invocation

2.4 Lecture Summary

2.4 Multicast Sockets

Lecture Summary: In this lecture, we learned about *multicast sockets*, which are a generalization of the standard socket interface that we studied earlier. Standard sockets can be viewed as *unicast* communications, in which a message is sent from a source to a single destination. *Broadcast* communications represent a simple extension to unicast, in which a message can be sent efficiently to all nodes in the same local area network as the sender. In contrast, *multicast sockets* enable a sender to efficiently send the same message to a specified set of receivers on the Internet. This capability can be

very useful for a number of applications, which include news feeds, video conferencing, and multi-player games. One reason why a 1:n multicast socket is more efficient than n 1:1 sockets is because Internet routers have built-in support for the multicast capability.

In recognition of this need, the Java platform includes support for a MulticastSocket class, which can be used to enable a process to join a *group* associated with a given MulticastSocket instance. A member of a group can send a message to all other processes in the group, and can also receive messages sent by other members. This is analogous to how members of a group-chat communicate with each other. Multicast messages are restricted to *datagrams*, which are usually limited in size to 64KB. Membership in the group can vary dynamically, i.e., processes can decide to join or leave a group associated with a MulticastSocket instance as they choose. Further, just as with group-chats, a process can be a member of multiple groups.

Optional Reading:

1. Documentation on MulticastSocket class

2. Wikipedia article on Multicast

2.5 Lecture Summary

2.5 Publish-Subscribe Pattern

In this lecture, we studied the *publish-subscribe pattern*, which represents a further generalization of the multicast concept. In this pattern, *publisher* processes add messages to designated *topics*, and *subscriber* processes receive those messages by registering on the topics that they are interested in. A key advantage of this approach is that publishers need not be aware of which processes are the subscribers, and vice versa. Another advantage is that it lends itself to very efficient implementations because it can enable a number of communication optimizations, which include *batching* and *topic partitioning* across *broker* nodes. Yet another advantage is improved reliability, because broker nodes can replicate messages in a topic, so that if one node hosting a topic fails, the entire publish-subscribe system can continue execution with another node that contains a copy of all messages in that topic.

We also studied how publish-subscribe patterns can be implemented in Java by using APIs available in the open-source *Apache Kafka* project. To become a publisher, a process simply needs to create a KafkaProducer object, and use it to perform send() operations to designated topics. Likewise, to become a consumer, a process needs to create a KafkaConsumer object, which can then be used to subscribe to topics of interest. The consumer then performs repeated poll() operations, each of which returns a batch of messages from the requested topics. Kafka is commonly used as to produce input for, or receive output from, MapReduce systems such as Hadoop or Spark. By storing Kafka messages as key-value pairs, data analytics applications written using MapReduce programming models can seamlessly interface with Kafka. A common use case for Kafka is to structure messages in a topic to be as key-value pairs, so that they can be conveniently used as inputs to, or outputs from, data analytics

applications written in Hadoop or Spark. Each key-value message also generally includes an *offset* which represents the index of the message in the topic. In summary, publish-subscribe is a higher-level pattern than communicating via sockets, which is both convenient and efficient to use in situations where producers and consumers of information are set up to communicate via message groups (topics).

Optional Reading:

1. Wikipedia article on the Publish-Subscribe pattern

2. Wikipedia article on the Apache Kafka project

Mini Project 2: File Server

---

miniproject_2.zip

Project Goals and Outcomes

This week we learned about communicating between Java Virtual Machines (JVMs) using sockets, and how high-level Java APIs make this process much simpler for programmers by offering stream-based communication APIs.

In this mini-project, you will gain experience working with Java Sockets by implementing a simple, stripped down file server that responds to HTTP requests by loading the contents of files and transmitting them to file server clients.

HTTP (or the Hypertext Transfer Protocol) was originally developed to support the communication of linked documents from early web servers to early web clients. It establishes a uniform protocol (or language) that all computers must be able to speak over the network in order to request HTTP documents. That protocol is based on a few fundamental commands, one of which is the GET command. Semantically, a GET request issued by an HTTP client to an HTTP server indicates that the client would like to retrieve a piece of read-only information from the server. For example, one common usage of GET is to request files from a file server by passing the path to the desired file. If a client were requesting a file called /path/to/file, the HTTP request would be a string starting with:

GET /path/to/file HTTP/1.1

This string indicates the command (GET), the resource requested (/path/to/file), and the version of the HTTP protocol being used (1.1). In general, many other lines would follow this line to form a complete HTTP request with other metadata such as information on the client performing the request.

An HTTP response to the above GET might look something like the following:

HTTP/1.0 200 OK\r\n

Server: FileServer\r\n

\r\n

FILE CONTENTS HERE\r\n

The first line of this response indicates that the file was found successfully. The second line identifies the type of server being spoken to. The third line serves as a separator between the HTTP header and the body of the request on the fourth line, the contents of the file itself. Note that the \r and \n are special-purpose new line characters required by the HTTP protocol.

Of course, like any robust software system, HTTP must also integrate an error model that allows a server to indicate to clients when something has gone wrong. One of the most common HTTP error codes (and likely one that you have personally run into) is 404 Not Found. 404 Not Found indicates that the resource requested by a GET was not found on the server it was requested from. In the case of a file server, this would indicate that the client requested the contents of a file that did not exist on the server. An example 404 response is shown below:

HTTP/1.0 404 Not Found\r\n

Server: FileServer\r\n

\r\n

In this mini-project, you will implement a simple file server that responds to HTTP GET commands with the contents of the requested file. In the case of a client requesting a file that does not exist, you will return a 404 Not Found error instead.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the project files using the gray button labeled miniproject_2.zip at the top of this description, you should see the project source code file at

miniproject_2/src/main/java/edu/coursera/distributed/FileServer.java

and the project tests in

miniproject_2/src/test/java/edu/coursera/distributed/FileServerTest.java

It is recommended that you review the demo video for this week before starting this assignment, in which Professor Sarkar works through a similar example.

Project Instructions

Your modifications should be made entirely inside of FileServer.java. You should not change the signatures of any public or protected methods inside of FileServer.java, but you can edit the method bodies and add any new methods that you choose. We will use our copy of FileServerTest.java in the final grading process, so do not change that file or any other file except FileServer .java.

Your main goals for this assignment are as follows:

1. Inside FileServer.java, implement the run method to use the provided ServerSocket object and root directory to implement the basic HTTP file server protocol described above. There are four helpful TODOs in FileServer.java to help guide your implementation.

You are provided with a proxy filesystem class called PCDPFilesystem which you will use to read files requested by file server clients. The only PCDPFilesystem API you need to be aware of is called readFile, which accepts a PCDPPath object representing the path to a file you would like the contents of. readFile returns the contents of the specified file if it exists. Otherwise, it returns null.

The PCDPPath object has a single constructor which accepts the absolute path to a file as a String. Hence, you can construct a PCDPPath object after parsing the requested file from an HTTP GET request.

Project Evaluation

Your assignment submission should consist of only the FileServer.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Please give it a few minutes to complete the grading. Note that for all assignments in this course you are free to resubmit as many times as you like. Once it has completed, you should see a score appear in the “Score” column of the “My submission” tab based on the following rubric:

- 10% - correctness of testTermination
- 10% - correctness of testFileGet
- 10% - correctness of testFileGets
- 10% - correctness of testNestedFileGet
- 20% - correctness of testDoublyNestedFileGet
- 10% - correctness of testLargeFileGet
- 10% - correctness of testMissingFileGet
- 20% - correctness of testMissingNestedFileGet