CS11 – Advanced Java

Winter 2011-2012
Lecture 6

Today's Topics

- Java object serialization
- Networking options!
  - TCP networking
  - UDP networking
  - Remote Method Invocation
  - Applicable uses of each

- Networked Boggle!

Serializing Objects

- Often need to convert between objects and a byte-sequence
    - Called “object serialization”
    - Converting from byte-sequence back to object is called “deserialization”
- Two main scenarios for object serialization:

1. Saving object state to persistent storage
    - Convert object into a byte-sequence, then save it to a file
    - Later, read byte-sequence from file and recreate the object
2. Sending an object to another JVM or computer
    - Convert object to byte-sequence, then send byte-sequence to the other JVM
    - Other JVM converts byte-sequence back into the object

Java Serialization

- Provided by two stream implementations:
  - **`java.io.ObjectOutputStream`**
    - Constructor takes an **`OutputStream`** argument
    - **`public void writeObject(Object obj)`**
    - Among many other capabilities, converts an object to a byte-sequence describing both type and state details
  - **`java.io.ObjectInputStream`**
    - Constructor takes an **`InputStream`** argument
    - **`public Object readObject()`**
    - Converts a byte-sequence back into an object, using type and state data in the byte-stream

Java Serialization Protocol

- Java serialization protocol is very complete and rich
- Each object's type information is included
  - First time a specific type is sent, type details are included
    - Class-name, field names and types
    - If class has parent-classes or class-fields, the type info is sent for those types as well
    - Each type is assigned an ID
  - Subsequently, just the type-ID is sent with an object
- Necessary overhead for a *generic* serialization mechanism
  - A custom-built serialization mechanism would be faster and generate smaller results…
  - …but, Java's serialization mechanism is *very* easy to use.

Using Java Serialization (1)

- Not all objects can be serialized!
  - Only ones that implement **`java.io.Serializable`**
  - Many Java collections, arrays, etc., are serializable
- **`Serializable`** is a tag interface
  - Specifies whether a class can be serialized or not
- If a base-class implements **`Serializable`**, derived classes are also serializable
- If a base-class doesn't implement **`Serializable`**, derived classes *can* implement **`Serializable`**…
  - But, derived classes must specially handle base-class serialization and deserialization. (Ugh!)

Using Java Serialization (2)

- Serializable objects must contain serializable data
  - All fields in the object must be serializable
  - All primitive types are serializable
- Any object fields must also be of a serializable type
  - Arrays are serializable if all elements are serializable
  - Most collection classes in **`java.util`** are serializable
- If an object (or its contents) isn't serializable:
  - A **`NotSerializableException`** is thrown when **`ObjectOutputStream.writeObject()`** is called

Serializing Objects

- Objects almost always refer to other objects
- Java serialization reads and writes *graphs* of objects
  - Simple graph-traversal algorithm
  - When an object is written to the stream, serializer assigns it a unique ID
    - Both the object’s ID and its data are written to the stream
  - Next time the object is encountered, serializer writes only the object’s ID
- Scenario:
  - You create an object and write it to an object-stream.
  - Then you change it, and write it to the object-stream again.
  - What does the stream’s reader see?

Serializing Objects (2)

- Scenario:
  - You create an object and write it to an object-stream.
  - Then you change it, and write it to the object-stream again.
- What does the stream's reader see?
  - Unfortunately, reader gets two copies of the original object
  - Changes aren't reflected in the stream, since Java serializer only looks at the object reference, not its state
- **`ObjectOutputStream`** has a **`reset()`** method
  - Resets <u>all</u> internal serializer state
  - Necessary when resending changes to the same object
  - Also generates <u>big</u> overhead as all type details are resent!

Transient Fields

- Serializable objects don’t have to serialize *all* fields
- Fields can be marked transient
  - Transient fields are not serialized or deserialized

    ```
    public class ComputeTask implements Serializable {
      private transient File outputFile;
      ...
    }
    ```

  - **`outputFile`** is not serialized or deserialized
    (A good thing: **`java.io.File`** is not serializable!)
- Exposes Java’s roots as a networking-friendly language: explicit language support for serialization

Serialization Strengths and Weaknesses

- Serialization is great for sending objects across a network
  - The serialized version isn't around for very long!
- Not so great for persistent storage of objects
  - A common scenario:
    1. Serialize objects to a file
    2. Add new fields/methods to the serializable classes
    3. Try to deserialize your data: Exception!
  - Problem: the <u>storage format</u> changed

Serial Version UIDs

- Java assigns a "serial version UID" to your class, based on its fields and field-types
  - Version ID is stored with object in output-stream data
  - Calculation method can vary from JVM to JVM!
- If class changes, serial version UID also changes
  - Deserializer reports an error if data-stream's serial version UID doesn't match the class' current version UID
- Can find out a class' current serial version UID
  - **`serialver`** ***`classname`***
  - Example:

```
% serialver MyClass
MyClass:    static final long serialVersionUID = -1993449670359138314L;
```

Final Serialization Details

- Can customize object-serialization in <u>many</u> ways
  - Especially important when supporting multiple serialized versions of your objects!
  - Can also look at **`java.io.Externalizable`** interface for complete control over serialization of object's data
- Serialization can open up security issues!
  - Private fields are serialized too – easy to access or change directly in the raw data stream
  - Easy to construct a byte-stream, then deserialize into an object that you shouldn't have access to
  - Must take these issues into account in secure systems!
    - Don't allow serialization, or encrypt/sign serialized data

Serialization Documentation

- Java serialization is very well documented by Sun
  - http://java.sun.com/javase/6/docs/technotes/guides/serialization/index.html
- Can actually look at Sun's implementation of serialization and deserialization
  - Source-code for Java API implementation included in JDK
- Effective Java also has a section on serialization
  - Joshua Bloch
  - See Chapter 10 (Serialization) for details

Networked Applications

- Networked application design:
    - Many communication tools to choose from!
- Can implement communications directly, using TCP/IP or UDP
- Can use a higher-level communication mechanism, like RMI
    - Remote Method Invocation
- Many other networking libraries available, too
- Best tool for the job depends on what the application is doing

TCP/IP Networking

- TCP = Transmission Control Protocol
- IP = Internet Protocol
  - TCP is layered on top of IP
  - Usually just called TCP
- Reliable, ordered, stream-based protocol
- Useful when data *must* be sent and received reliably
- Protocol imposes extra overhead, so it is a little slower than max network capabilities
  - This can be tuned in several ways, based on actual usage

Java TCP Communication

- TCP communication *requires* a connection
  - Another benefit: you know when your peer disconnects!
- Client uses **`java.net.Socket`** to connect
  - Hostname and port must be specified
- Server uses **`java.net.ServerSocket`** to accept connections
  - **`accept()`** method must be called for every client that connects
  - Returns a **`Socket`** object that can be used to talk to the client
- **`Socket`** provides streams for communication

UDP Networking

- Universal Datagram Protocol
  - Unreliable, unordered, message-based communications
  - Packets might arrive in different orders
    - Sender sends P_1 then P_2
    - Receiver receives P_2 then P_1
  - A packet might arrive multiple times
  - A packet may not arrive at all
- Messages are called “datagrams”
- Good choice when data’s relevance expires quickly
- UDP also provides broadcast and multicast features

Java UDP Networking

- **`java.net.DatagramSocket`** provides UDP communication
  - Very different lifecycle from TCP communications!
- When socket is created:
  - Socket can be bound to a local address and/or a port
  - Socket may be unbound – not associated with any address
- Before sending or receiving a datagram (a packet), socket must be bound to a local address
- Socket doesn't *have to* connect to a remote host before sending a datagram to that host
  - UDP is a <u>connectionless</u> protocol
  - Can connect a socket to a specific host, but then can only send/receive with that host

Datagrams

- **`DatagramPacket`** represents datagrams in Java
- A datagram contains (among other things):
  - The data being sent
  - The source address for the datagram
  - The destination address for the datagram
- Datagrams are routed entirely based on their internal information
  - This is why UDP doesn’t require connections
- A program receiving datagrams can determine what hosts/ports the datagrams are from
  - Can send a response back to each sender, even in absence of an actual connection with the sender

Datagram Data (1)

- The actual data in the datagram is just a byte-array
  - Your application specifies the data to send or receive
  - The “application-layer protocol”
- Can use **`java.io.ByteArrayOutputStream`** to generate datagram data
  - Wrap it with a **`DataOutputStream`** to write all primitive data-types
  - Wrap with **`ObjectOutputStream`** to write primitive types and objects
- Then, use **`java.io.ByteArrayInputStream`** to reconstitute datagram data
  - Again, wrap it with an appropriate stream to do conversions

Datagram Data (2)

- **`ByteArrayOutputStream`** has **`toByteArray()`** method
  - Makes a copy of the internal data! SLOW.
  - Subclass **`ByteArrayOutputStream`** to provide access to internal **`buf`** and **`count`** fields
  - Or, provide a **`copyToByteArray()`** method that lets the caller provide an array to copy into.
    - Much safer approach.
- **`ByteArrayInputStream`** needs similar trickery
  - Provide methods to store new data into the stream, and reset its position, etc.
- Avoid creating extra objects per packet, if possible!

Other UDP Notes

- UDP broadcast usually only works on local subnet
  - Routers don’t usually forward broadcast packets (for obvious reasons)
- UDP multicast is also unreliable, unordered
  - Routers don’t always support this protocol
- Routers may decide to drop UDP packets
  - If network is congested, routers drop larger packets first!
  - Keeping packets to under 1.5KB is usually safest
    - Maximum Transmit Unit (MTU) = 1500B for Ethernet, 1492B for PPPoE/DSL

Byte-Ordering Issues

- Byte-order is very important in networking protocols
  - Different architectures store multibyte values in different byte-orders
  - Little-endian: higher addresses store most significant bits
  - Big-endian: lower addresses store most significant bits
- Programs typically convert to “network byte-order” before sending data over the network
  - Network byte-order is big-endian
  - Ensures a common byte-ordering across different platforms
- Java **`DataInput`** and **`DataOutput`** interfaces specify big-endian order, so no concerns here!

Remote Method Invocation

- Much higher-level networking mechanism
- A program exposes objects that can be called from remote hosts
  - Called server objects, or remote objects
  - Each remote object has its own string name or path
- Client requests access to a remote object, by name
  - Client has to connect to machine where remote object is
  - Client gets back a stub: it exposes exact same interface, but is local to the client
- Client calls methods on the stub
  - Arguments are serialized and sent to the remote object
  - Return-value (or exception) is serialized and sent back

RMI Mechanics

- Each remote object has its own name
  - An RMI registry (of some form) must be available
  - Registry is usually a separate program from JVM
    - Can also start a registry within the server program
  - Server objects must be registered before use
  - Clients contact registry to obtain a remote object
- Stub is client's "view" of the remote object
  - Stub provides same API as remote object
  - Responsible for dispatching calls over the network and receiving the response for the client

Remote Objects

- All remote objects are exposed via interfaces
  - Interfaces are derived from **`java.rmi.Remote`**
  - Remote interfaces define the methods that can be invoked from other machines
- Interface methods must say they can throw **`java.rmi.RemoteException`**
  - Many possible failures in remote method invocation!
  - The interface implementation itself usually doesn't throw **`RemoteException`**
  - Another step in the invocation process might throw it
- All arguments and return-values must be serializable
  - Your remote interface can specify exceptions too…
  - <u>All</u> exceptions are serializable (**`Throwable`** is serializable)

RMI Protocols (1)

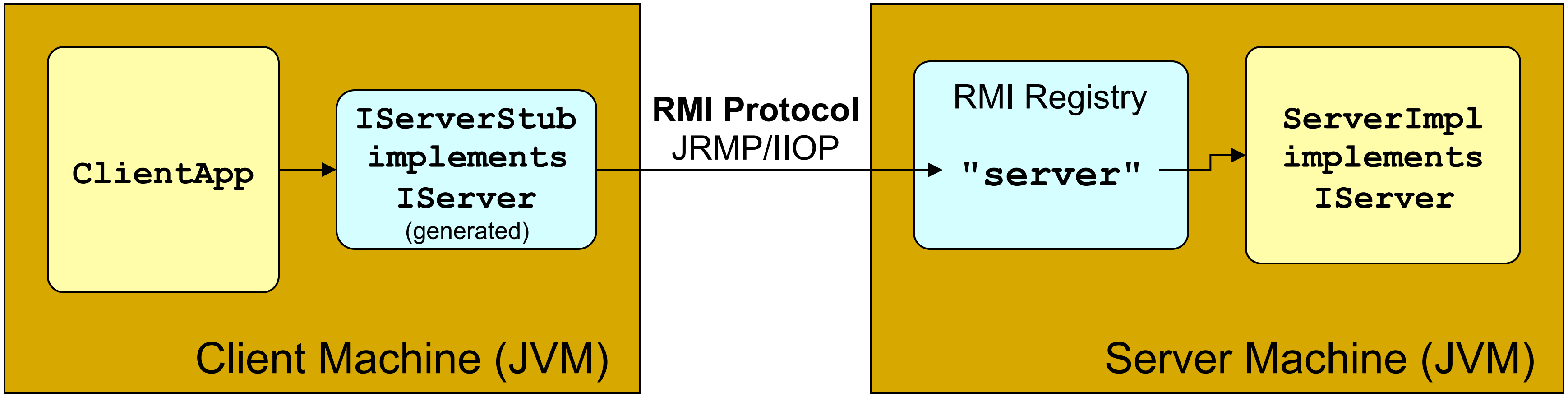
- RMI-JRMP (aka “RMI over JRMP”)
  - Java Remote Messaging Protocol
  - Calls between Java objects only
  - Easy, and appropriate for most pure-Java applications
- RMI-IIOP (aka “RMI over IIOP”)
  - Internet Inter-ORB Protocol
  - CORBA: Common Object Request Broker Architecture
    - Object Management Group (http://www.omg.org)
  - ORB: Object Request Broker
  - Can call Java objects from (possibly non-Java) clients
  - Java clients can call (possibly non-Java) remote objects
  - Often necessary for large-scale enterprise apps
    - (Support integration with legacy software or external systems)

RMI Protocols (2)

- SOAP
  - Simple Object Access Protocol
  - XML-based RMI operations, performed over HTTP
    - “Web-services”
  - Also uses many concepts and classes from Java RMI
  - Apache Axis2: http://ws.apache.org/axis2/
- Choose RMI protocol based on application’s needs
  - JRMP is best for “Pure Java” applications, and is default
  - IIOP is best for integrating disparate systems (possibly in different languages) with Java
  - SOAP is best for web-application systems, and more firewall-friendly RMI interactions (can use HTTP port 80)

RMI Components

- RMI Interactions, as of Java 1.5+



- Before Java 1.5:
  - Server also had a “skeleton” class for each remote object
  - Manually generated stubs and skeletons with **`rmic`** tool
  - Still need to use **`rmic`** for interfacing with Java 1.4 or older RMI/JRMP systems, or RMI/IIOP systems of any version

Tag Interfaces

- Already discussed “Constant Interfaces”
  - Java interfaces can include constant declarations
  - Constant Interfaces <u>only</u> contain constants; no methods!
  - Discouraged because they don’t specify a set of behaviors
- Another common Java pattern: Tag Interfaces
  - Also called “marker interfaces”
  - An interface with no methods that can be used to tag sub-interfaces or objects
  - No constants either; the interface is completely empty
  - Indicates that the object supports special usage scenarios, but object itself doesn’t provide them

Tag Interfaces (2)

- Example: **`java.lang.Cloneable`**
- From API docs:
  - A class implements the **`Cloneable`** interface to indicate to the **`Object.clone()`** method that it is legal for that method to make a field-for-field copy of instances of that class.
- **`Cloneable`** doesn't declare any methods!
  - **`java.lang.Object`** has an implementation of **`clone()`**
  - Implementation throws **`CloneNotSupportedException`** if **`clone()`** is called on a non-**`Cloneable`** object
- Tag interfaces specify behavior… sort of…

Tag Interfaces and Annotations

- Tag interfaces were included in Java 1.0
  - …back when annotations simply didn’t exist
  - Needed a way to annotate objects, using then-extant Java language features
- With Java 1.5 annotations, tag interfaces *could* be phased out
  - For example:
    - **`@Cloneable`** annotation for cloneable objects
    - **`@Serializable`** and **`@Transient`** annotations
  - No such annotations exist… yet…

Tag Interfaces and RMI

- Tag interfaces related to RMI:
- **`java.io.Serializable`**
  - Used to tag objects
  - “This object can be converted to/from a byte-stream using the Java object-serialization mechanism.”
- **`java.rmi.Remote`**
  - Used to tag *sub-interfaces* derived from it
  - “Sub-interfaces deriving from **`Remote`** can be called from other processes or machines.”
  - “Objects implementing sub-interfaces of **`Remote`** can be exposed in an RMI registry.”

Building Distributed Systems

- Different network communications options!
  - Different features, strengths, weaknesses
- Want to pick the right tool for the job
  - Some communications options simply don’t provide the features you need
  - Sometimes performance is an issue
  - Maximize the results of your efforts
    - “Constructive laziness”
    - Use other people’s hard work on these problems.

Networking Choices: UDP

- UDP is good for:
  - Fast, unreliable communications
    - e.g. position updates in a networked game
  - Clever networking tricks and functionality
    - Broadcast to subnet – great for auto-discovery of peers
    - Multicast communications
  - Can apply to client-server or peer-to-peer models
  - Great for sending event notifications
- If you don’t *definitely* need UDP, consider using TCP instead (with proper configuration)

Networking Choices: TCP

- TCP is good for:
  - Reliable, stream-based communications
    - Slower than UDP, but can definitely be fine-tuned for your system’s needs!
  - Can be applied most easily to client-server model
    - Peer-to-peer model is perfectly feasible too, but requires careful design
  - *Great* for moving large amounts of data around
  - Also good for control messages or events that *must* reach their destination
  - Client or server can send data anytime

Networking Choices: RMI

- RMI is good for:
  - Constructing distributed systems with functionality exposed entirely as method-calls
  - Avoid the hassle of creating a networking protocol
  - Entirely request/response-based applications!
    - Servers cannot fire events back to clients
    - Clients can periodically poll server for notifications
      - Expensive from a networking standpoint, and slow.
    - Client could also expose remote objects and a registry
      - Very complicated! But sometimes this is acceptable.
    - TCP is much better for asynchronous event passing

Networked Boggle!

- This week's lab:
  - Get Boggle server up and running with your client!
- Most of the server implementation is provided
- You have to:
  - Get client and server to talk via RMI
    - Server **`main()`** method exposes interface via RMI registry
    - Client **`main()`** method retrieves server's remote interface
  - Update client to call server to start and end rounds
    - Update your controller
  - Implement the game-scoring portion of the server
    - Find each client's unique words, score each client's words

Boggle Server

- Boggle app will use RMI for communications
- The hard part:
  - *How to coordinate players who join the game at different times??*
- A simple solution:
  - Boggle server interface has two methods:
    - **`startGame()`**
    - **`gameOver()`**
  - When a client calls server's **`startGame()`** method, server doesn't allow the call to return until the next round starts
    - Call to **`startGame()`** <u>blocks</u>, until next game actually starts
  - Different RMI calls to the server occur on different threads
  - Server logic manages the incoming calls to implement this

Boggle Server (2)

- Boggle server code you get:
  - Code that handles RMI calls from multiple clients
  - Code that handles players that join in the middle of a Boggle round
  - Other classes for managing game state
- Ready for use in an RMI client/server system, but you will have to get it working
  - Make sure everything is serializable
  - Make sure server interface conforms to “remote interface” requirements
  - Get server to expose its remote interface in an RMI registry
  - Get client to connect to the server!