LECTURE: MODULARITY, SERVICES, AND API-DRIVEN APPLICATIONS

CHANDRA KRINTZ – UCSB CS263
CHANGES IN PROGRAMMING LANGUAGES AND RUNTIMES

• From last time: External trends and influences
  • Hardware/architecture evolution
    • Low cost, HPC, memory-rich, multicore, virtualization
    • Moore’s law (Dennard scaling is dead)
  • Distributed computing
    • Remote execution, multi-tasking, containerization, cloud
  • The people who are developing applications/software
    • Productivity programmers vs specialists/experts
    • Focus on enabling productivity
• External trends and influences
  • Have changed language/runtime support

• Your project in this class takes advantage of each:
  • High-level, OO, garbage collected language: Java
  • Modular development
    • Interoperable and communicating (RPC, data transfer)
      • Language-specific and Language-independent (JSON)
      • Web service APIs - language independent interoperation (REST)
  • Leveraging support tools (JAX-RS)
  • Deployment and programming model of popular cloud platform (Google App Engine)
OUTLINE

• External trends and influences
  • **Have changed language/runtime support**

• Your project in this class takes advantage of each:
  • **High-level, OO, garbage collected language: Java**
  • Modular development
    • Interoperable and communicating (RPC, data transfer)
      • Language-specific and Language-independent (JSON)
      • Web service APIs - language independent interoperation (REST)
    • Leveraging support tools (JAX-RS)
  • Deployment and programming model of popular cloud platform (Google App Engine)
CHANGES IN PROGRAMMING LANGUAGES

• Object oriented
  • Encapsulation, reuse, complexity management

• Portable (write-once, run-anywhere), mobile

• Facilitating modularity and isolation
  • Independent and separate development/testing/deployment
  • Managed runtimes contain faults, sandbox execution

• Garbage collected
  • For portability, error reduction, memory/type safety
OUTLINE

• External trends and influences
  • Have changed language/runtime support

• Your project in this class takes advantage of each:
  • High-level, OO, garbage collected language: Java
  • Modular development
    • Interoperable and communicating (RPC, data transfer)
      • Language-specific and Language-independent (JSON)
      • Web service APIs - language independent interoperation (REST)
    • Leveraging support tools (JAX-RS)
  • Deployment and programming model of popular cloud platform (Google App Engine)
CHANGES IN SOFTWARE DEVELOPMENT

• Each component in its own managed runtime
  • Fault isolation
  • Separate concerns
  • Can execute anywhere
    • Same/different machine
  • Optimize/specialize runtime

• Emphasis moves to
  • Locating/naming
  • Coordination
    • Data transfer and persistence
  • Communication
    • Interprocess communication (IPC)

JVM=VM=MRE=Runtime = execution engine
CROSS-RUNTIME COMMUNICATION & COORDINATION

• Java interprocess communication (IPC)
  • Data exchange formats
    • Language specific serialization
    • Language-independent serialization
  • RPC, messaging
    • Invocation of procedures located in another address space, or on different machine
      • Copy semantics: caller & callee operate on their own copies of the data
    • Java RMI (requires support for naming, eg JNDI, registries)
• Data persistence
  • JDO, JDBC
• Networking
  • HTTP, TCP/IP sockets
JAVA OBJECT SERIALIZATION

• Serialization is the process of transforming an in-memory object to a byte stream so that it can be transferred over a network.

• Deserialization is the inverse process of reconstructing an object from a byte stream to the same state in which the object was previously serialized.

• “Serializing out” and “serializing in” are also used.
JAVA (DE-)SERIALIZATION CODE

ObjectOutputStream oos =
    new ObjectOutputStream(new FileOutputStream("sname"));
oos.writeObject ( new Date() );
oos.close();

ObjectInputStream ois =
    new ObjectInputStream(new FileInputStream("sname"));
Date d = (Date) ois.readObject();
ois.close();
Java Serialization Basics

- The requirements for serialization are straightforward:
  - Only class instances rather than primitive types can be serialized.
  - For an object to be serializable, its class or some ancestor must implement the `empty Serializable` interface.
  - An empty interface is called a *marker interface*.

- The syntax for serialization is straightforward:
  - An object is serialized by writing it to an `ObjectOutputStream`.
  - An object is deserialized by reading it from an `ObjectInputStream`.

- The classes must be the same in both endpoint JVMs.
OBJECT GRAPHS

• If an object has references to other objects or arrays, the entire *object graph* is serialized when the object is serialized.
  • Graph consists of the object and those to which the object has direct or indirect references
  • If any of these objects do not implement the `Serializable` interface, the runtime throws a `NotSerializableException`
OBJECT GRAPHS

• If an object has references to other objects or arrays, the entire object graph is serialized when the object is serialized.
  • Graph consists of the object and those to which the object has direct or indirect references
  • If any of these objects do not implement the Serializable interface, the runtime throws a NotSerializableException

• If a serializable class C has a nonserializable superclass S, instances of C still can be serialized if S has a non-private default (no-argument) constructor
  • We need to be able to construct C’s parent (S) parts
  • S’s default constructor is invoked automatically during deserialization to construct “S-part” of the deserialized object
SERIALIZATION AND PRIMITIVE TYPES AND OTHER NOTES

- Primitive types cannot be serialized or deserialized.
  - `ObjectOutputStream` implements the `DataOutput` interface, which declares methods such as `writeInt` to write primitive types to streams.
  - `ObjectInputStream` implements `DataInput` for reading primitive types.
- The same object should not be repeatedly serialized to the same stream.
- A class should not be redefined in between the serialization and deserialization of its instances.
- To prevent serialization of an object, a class throws a `NotSerializableException` in its private callback `writeObject` method.
**TRANSIENT AND STATIC FIELDS & CUSTOMIZATION**

- **transient** fields are not impacted by serialization
  - Deserialization: **transient** fields restored to their **default** values
- **static** fields are not serialized

- Serialization and deserialization can be customized by providing **private** callback methods named `writeObject` and `readObject`, respectively.

- **Externalizable** interface gives dev complete control
  - Must implement `writeExternal()` and `readExternal()`
CROSS-RUNTIME COMMUNICATION & COORDINATION

• Java interprocess communication (IPC)
  • Data exchange formats
    • Java (de-)serialization

• RPC, messaging
  • Invocation of procedures located in another address space, or on different machine
    • Copy semantics: caller & callee operate on their own copies of the data
  • Java RMI (requires support for naming, eg JNDI, registries)
    • Language specific RPC
JAVA RPC: REMOTE METHOD INVOCATION

- A Java-only solution to RPC
  - Underlying wire protocol is object serialization: not open standard
  - An alternative to sockets (tedious to implement, error prone)
- Instantiate an object on another machine or in other JVM
- Invoke methods on the remote object
JAVA RMI DEVELOPMENT STEPS

- Create the Interface to the server
- Create the Server
- Create the Client
- Compile the Interface (javac)
- Compile the Server (javac)
- Compile the Client (javac)
- Generate Stubs (rmic)
The RMI Registry is a naming service provided with the JDK as a teaching tool or for a small number of Remote Objects.

- Uses port 1099 as its default port
- Can be considered to be a reference implementation
- Runs out of steam above a 100 objects
- Runs on the same machine as the remote object

Use another naming service

- J2EE uses JNDI and Directory Services to provide a more robust naming service
- Silverstream uses JNDI with its own ServiceProvider and repository for RMI
CROSS-RUNTIME COMMUNICATION & COORDINATION

- Java interprocess communication (IPC)
  - Data exchange formats
    - Language specific serialization

- RPC, messaging
  - Invocation of procedures located in another address space, or on different machine
    - Copy semantics: caller & callee operate on their own copies of the data
  - Java RMI (requires support for naming, e.g., JNDI, registries)

- Data persistence
  - Long-term storage of information after program exit
    - File I/O
    - JDO, JDBC
PERSISTENCE FOR JAVA DATA / OBJECTS

• File IO

• **Serialization**. Simple API; no support for query, sharing, transactions, or partial read or update. In addition, it is not suitable for large-capacity data storage.

• **Java DataBase Connectivity (JDBC) software**. Full access to SQL DB functionality. Developer explicitly manages the values of fields and their mapping into relational tables.

• **Enterprise JavaBeans (EJB) architecture Container Managed Persistence (CMP)**. J2EE component model; provides a portable object persistence service to EJB containers; not general purpose.

• **JDO API**. A standard/library approach for achieving object persistence in Java technology that uses a combination of XML metadata & bytecode manipulation.
JAVA RPC AND DATA EXCHANGE

• Benefits from these technologies
  • Programmer productivity (abstraction, portability, isolation, copy semantics)

• Limitations
  • Costly operations to serialize data
  • Network communication
  • Single language (Java only for everything)
    • Modern applications are multilanguage and employ web 2.0
      • Presentation layer: Javascript, Ruby, Java, Python
      • Server-side logic: PHP, Perl, Java, Python, Ruby
      • Computation: MapReduce streaming (multi-language)
      • Database, key-value store: C++, Java, + query languages
CROSS-LANGUAGE DATA EXCHANGE AND RPC

• Python, Javascript, Perl, PHP, Ruby, Java, C/C++, .Net, ...

• **Language-independent** process communications
  • Data exchange formats
    • XML, Google Protocol Buffers, JSON

• RPC, messaging
  • CORBA, Thrift, HTTP/s, REST, SOAP, RPC, COM, SIP, SWIG
  • For more than just web services: Map-Reduce (MR), MR-streaming, MPI
JAVASCRIPT OBJECT NOTATION (JSON)

- Language independent: readers/writers via libraries
- Lightweight data interchange format
  - Compared to XML
    - Simple text format (curly braces, name at start only)
  - Easy for humans to read and write (self describing like XML, i.e. values are named)
    - Collections of key/value pairs or ordered list of values
    - Can’t use Javascript reserved words
    - Google protocol buffers are smaller/faster but binary
  - Easy for machines to parse and generate (data is directly accessible)
    - Values are typed (XML data is typeless)
    - Includes arrays (i.e. each element is nameless, XML doesn’t)
CROSS-LANGUAGE RPC: THRIFT

• From Facebook engineers
• Cross-language/process communications technology

• Distributed, transparent **cross-language** RPC framework
CROSS-LANGUAGE RPC: THREIF

• From Facebook engineers
• Cross-language/process communications technology

• Distributed, transparent cross-language RPC framework
• Language-neutral interface specification
  • Stub support for clients and servers
    • In different languages

• Simple architecture
• Limited in terms of generality (only static binding)
  • Changing interfaces, implementations requires recompilation
• External trends and influences
  • Have changed language/runtime support

• Your project in this class takes advantage of each:
  • High-level, OO, garbage collected language: Java
  • Modular development
    • Interoperable and communicating (RPC, data transfer)
      • Language-specific and Language-independent (JSON)
      • Web service APIs - language independent interoperation (REST)
  • Leveraging support tools (JAX-RS)
  • Deployment and programming model of popular cloud platform (Google App Engine)
TRENDS IMPACTING SOFTWARE DEVELOPMENT

• Hardware/architecture evolution
  • Low cost, HPC, memory-rich, multicore, virtualization
  • Moore’s law (Dennard scaling is dead)

• Distributed computing
  • Remote execution, multi-tasking, containerization, cloud

• The people who are developing applications/software
  • Productivity programmers vs specialists/experts

• Apps built by composition of components
  • Components increasingly becoming web-accessible services
  • APIs: gateway to digital assets, business opportunities
    • Software and data as-a-service
    • Proliferation of APIs
  • Applications increasingly rely on remote web services
WEB APIS ARE EVERYWHERE
WEB SERVICE INTEROPERATION: SOAP

• Simple object access protocol (SOAP)
  • 1998 D. Winer and D. Box at Microsoft
    • Evolved to be very extensive and complex
  • An XML-based message format and protocol
  • Language and platform independent
  • Runs over HTTP/S and a variety of other protocols
  • Standards
    • UDDI – locate an API/service
    • SOAP – communication protocol
    • WSDL – describe a web service
    • Hundreds of others
  • Many tools and language support

• Still used heavily in industry (now considered legacy)
WEB SERVICE INTEROPERATION: REST

- SOAP is heavy-weight
  - Onerous development requirements
  - Too many standards
  - XML message format is verbose
  - Most new development now uses REST

- REST: Representational State Transfer
REST

• 2000 PhD Dissertation by Roy Fielding at UC Irvine
• Fast, flexible web service architecture
  1. Client-Server model
  2. Stateless
  3. Caching
  4. Uniform interface
  5. Layered design
  6. Code-on-demand

• Tightly coupled to HTTP
REST

- 2000 PhD Dissertation by Roy Fielding at UC Irvine
- Fast, flexible web service architecture
  1. Client-Server model
  2. Stateless
  3. Caching
  4. Uniform interface
  5. Layered design
  6. Code-on-demand

- Tightly coupled to HTTP
- Identification of “resources” with *uniform interfaces*
  - Manipulation of resources through representations
- Self-descriptive messages
AN EXAMPLE

- Consider a learning management system (e.g. GauchoSpace)
  - Resources: Student, Course, Instructor

- Each resource can be uniquely identified by a URL
  - Student whose perm no. is 123456
    • /students/123456
  - Course CS263
    • /courses/cs263
  - All courses
    • /courses

---

@student>
  <perm>123456</perm>
  <firstName>Jane</firstName>
  <lastName>Doe</lastName>
  <email>jdoe@umail.ucsb.edu</email>
  <major>CS</major>
  <year>sophomore</year>
</student>

{ "perm" : 123456,
  "firstName" : "Jane",
  "lastName" : "Doe",
  "email" : "jdoe@umail.ucsb.edu",
  "major" : "CS",
  "year" : "sophomore" }
EXAMPLE CONT'D.

- Representational state transfer over HTTP
  - Use different HTTP methods to pass different commands (GET, POST, PUT, DELETE)

Client

GET /students/123456

200 OK
  <student>
  ....
  </student>

Server
CLIENT AS A STATE MACHINE

- Initialized → Place Order → Order Placed
  - Order Placed → Review/Update
  - Order Placed → Make Payment → Paid
  - Order Placed → Order Reviewed

- Make Payment
- Review/Update

- Make Payment
HATEOAS

- Hypermedia as the engine of application state (HATEOAS)
- Server provides the client with navigation information so the client can find its way around the application
- Client application should perform state transitions based only on the information provided by the server at runtime
HATEOAS EXAMPLE

**Place Order**

- **Client**
- **Server (example.com)**

POST /order

201 Created
Location: [http://example.com/orders/order100](http://example.com/orders/order100)
Next: [https://example.com/pay/order100](https://example.com/pay/order100)

**Review/Update Order**

- **Client**
- **Server (example.com)**

PUT /order/order100

200 OK
Next: [https://example.com/pay/order100](https://example.com/pay/order100)

**Checkout Order**

- **Client**
- **Server (example.com)**

POST /pay/order100

200 OK
REST: COMMON MISTAKES

- Every HTTP-based service is a RESTful service
- REST can only be implemented in HTTP
- REST is a service development methodology

Things to remember
- Interoperability and reuse not specified via standards but encouraged via best practices
  - Requires careful work to maintain interoperability
- Similarly: No standard mechanisms for implementing security and other QoS requirements

- Many frameworks and libraries do the work for you
JAX-RS

• A standard Java API for developing RESTful services

• Implement your service in plain old Java and add some annotations to expose it as a RESTful service

• Supported by a variety of programming frameworks and runtimes
  • Jersey, Apache CXF, Apache Wink, IBM WebSphere, Oracle WebLogic, RESTEasy, RESTlet etc
CURL

• Powerful command-line tool for testing HTTP endpoints

• GET:
  • curl –v http://test.com

• POST (Form):
  • curl –v –X POST –d “key=value” http://test.com

• POST (XML):
  • curl –v –X POST –d @test.xml –H “Content-type: application/xml” http://test.com
• External trends and influences
• Have changed language/runtime support

• Your project in this class takes advantage of each:
  • High-level, OO, garbage collected language: Java
  • Modular development
    • Interoperable and communicating (RPC, data transfer)
      • Language-specific and Language-independent (JSON)
      • Web service APIs - language independent interoperation (REST)
    • Leveraging support tools (JAX-RS)
  • Deployment and programming model of popular cloud platform (Google App Engine)
TRENDS IMPACTING SOFTWARE DEVELOPMENT

- Hardware/architecture evolution
  - Low cost, HPC, memory-rich, multicore, virtualization
  - Moore’s law (Dennard scaling is dead)

- Distributed computing
  - Remote execution, multi-tasking, containerization, cloud

- The people who are developing applications/software
  - Productivity programmers vs specialists/experts

- Apps built by composition of components
  - Components increasingly becoming web-accessible services
  - APIs: gateway to digital assets, business opportunities

- Cloud platforms have emerged to simplify process
  - Automatically configure, deploy, manage apps, APIs, and services