CS 550:
Advanced Operating Systems
Remote Procedure Call & Remote Method Invocation & Web Services

Ioan Raicu
Computer Science Department
Illinois Institute of Technology

CS 550
Advanced Operating Systems
February 3rd, 2011
• Wrap-up of RPC
• Case study: Sun RPC
• Extended RPC
  – Lightweight RPCs
  – Asynchronous RPC
  – One-way RPC
• Remote Method Invocation (RMI)
  – Design issues
  – Case study: JAVA RMI
• Web Services
• Remember “performance” one of the most important requirements
• Performance depends on ?
• RPC Protocol (options)
  – connection vs connectionless oriented
  – standard vs. specialized
Implementation Issues

- Choice of protocol
  - Use existing protocol or design from scratch
  - Packet size restrictions
  - Reliability in case of multiple packet messages
  - Flow control
- Copying costs are dominant overheads
  - Need at least 2 copies per message
  - As many as 7 copies
One of the most widely used RPC systems
– Also known as Open Network Computing (ONC)

Originally developed by Sun, but now widely available on other platforms (including Digital Unix)

Sun RPC package has an RPC compiler (rpcgen) that automatically generates the client and server stubs.

RPC package uses XDR (eXternal Data Representation) to represent data sent between client and server stubs.

Has built-in representation for basic types (int, float, char)

Also provides a declarative language for specifying complex data types
Example: RPC Programming

1. Write RPC protocol specification file \textit{foo.x}
2. Write server procedure \textit{fooservices.c}
3. Write client application \textit{foomain.c}
There is a tool for automating the creation of RPC clients and servers.
The program rpcgen does most of the work for you.
The input to rpcgen is a protocol definition in the form of a list of remote procedures and parameter types.
Example: RPCGEN

4. `rpcgen -C foo.x`

- `foo_clnt.c` (client stubs)
- `foo_svc.c` (server main)
- `foo_xdr.c` (xdr filters)
- `foo.h` (shared header file)
5. `gcc -o fooclient foomain.c foo_clnt.c foo_xdr.c -lnsl`

- `foomain.c` is the client main() (and possibly other functions) that call rpc services via the client stub functions in `foo_clnt.c`
- The client stubs use the `xdr` functions.
6. gcc -o fooserver fooservices.c foo_svc.c foo_xdr.c -lrpcssvc -Lnsl

• fooservices.c contains the definitions of the actual remote procedures.
7. Copy the server fooserver to the remote machine, and run it in the background

8. Now you can call the remote procedure on a local machine

- Useful reference:
Many RPCs occur between client and server on same machine
  - use a lightweight RPC mechanism (LRPC)
Server $S$ exports interface to remote procedures
Client $C$ on same machine imports interface
OS kernel creates data structures including an argument stack shared between $S$ and $C$
1. Copy args
2. Trap to Kernel
3. Upcall
4. Execute procedure and copy results
5. Return (trap)
Other RPC Models

- **Asynchronous RPC**
  - Server can reply as soon as request is received and execute procedure later

- **Deferred-synchronous RPC**
  - Use two asynchronous RPCs
  - Client needs a reply but can’t wait for it; server sends reply via another asynchronous RPC

- **One-way RPC**
  - Client does not even wait for an ACK from the server
a) The interconnection between client and server in a traditional RPC

b) The interaction using asynchronous RPC
Deferred Synchronous RPC

- A client and server interacting through two asynchronous RPCs
Remote Method Invocation (RMI)

• RPCs applied to objects
  – Class: object-oriented abstraction; module with data and operations
  – Separation between interface and implementation
  – Interface resides on one machine, implementation on another

• RMIs support system-wide object references
  – Parameters can be object references
When a client binds to a distributed object, load the interface ("proxy") into client address space.

Server stub is referred to as a skeleton.
• Proxy: client stub
  – Maintains server ID, endpoint, object ID
  – Sets up and tears down connection with the server
  – Does serialization of local object parameters
  – In practice, can be downloaded/constructed on the fly

• Skeleton: server stub
  – Does deserialization and passes parameters to server and sends result to proxy
An object reference must contain enough information to allow a client to bind to an object

- Object reference include server ID, endpoint, and object ID

- Have a local daemon per machine that keeps track of the server-to-endpoint assignments

- Use a location server

- Include an implementation handle in the object reference
Static vs. Dynamic RMI

• Static invocation
  – Use predefined interface definitions
  – Require that the interfaces of an objects are known when client application is being developed

• Dynamic invocation
  – A method invocation is composed at runtime
  – An application selects at runtime which method it will invoke at a remote objects
• **Server**
  – Defines interface and implements interface methods
  – Server program
    • Creates server object and registers object with “remote object” registry

• **Client**
  – Looks up server in remote object registry
  – Uses normal method call syntax for remote methods

• **Java tools**
  – rmic: java RMI stub compiler
  – rmiregistry: java remote object registry
  – rmid: java RMI activation system daemon

• **Useful reference:**
  – http://java.sun.com/j2se/1.4/docs/guide/rmi/
Java RMI

- Java supports Monitors: synchronized objects
  - Serializes accesses to objects
- Options: block at the client or the server
  - Block at server
    - Can synchronize across multiple proxies
    - Problem: what if the client crashes while blocked?
  - Block at proxy
    - Need to synchronize clients at different machines
    - Explicit distributed locking necessary
“Web services” is an effort to build a distributed computing platform for the Web.

Yet another one!
Designing Web Services

• Goals
  – Enable universal interoperability
  – Widespread adoption, ubiquity: fast!
  – Enable (Internet scale) dynamic binding
    • Support a service oriented architecture (SOA)
  – Efficiently support both open (Web) and more constrained environments

• Requirements
  – Based on standards. Pervasive support is critical
  – Minimal amount of required infrastructure is assumed
    • Only a minimal set of standards must be implemented
  – Very low level of application integration is expected
    • But may be increased in a flexible way
  – Focuses on messages and documents, not on APIs
Web service applications are encapsulated, loosely coupled Web “components” that can bind dynamically to each other.
Web Services Summary

• Web Services are logically simple
  – Standard mechanisms for describing, discovering, and accessing services
  – Encourage loose coupling; service-oriented architecture

• Web Services are complex in practice
  – Due to the wide variety of interactions that can occur

• Broad adoption is encouraging.

• For more information
  – Web Services Architecture: http://www.w3.org/TR/ws-arch/