V22.0480-004
Web Services Architecture and Programming

Lecture 5
Remote Procedure Calls (cont’d)
Announcements

• Lab 1 due back on September 23rd
  – No extensions
  – Use the mailing list for questions/clarifications

• Hand-in instructions
  – Put your files in the following folder: D:\VSDev\{user-name}\Lab1
  – Send e-mail to both TAs and me with the lab write-up
  – E-mail should be received by 12:00 noon on the 23rd

• Lab 2 (XML-RPC) will be handed out on September 23rd, due back October 7th
(Review) Overall Structure of RPC

- Client process blocks for duration of the call
  - Just like in a local procedure call
  - Asynchronous RPC: early reply from server

- RPC package is at the session layer
  - Can work with different transports
    - Shared Memory, UDP or TCP
    - has to be specified at setup time

- Message passing completely hidden from programmer
Understanding RPC: Client and Server Stubs

- We want to make RPC’s look like local procedure calls
- Client stubs allow callers to make remote calls that look like local calls
- Server stubs allow callees to respond to remote calls as if they were from a local caller
Issue 1: RPC Parameter Passing

- Client and server stubs need to ensure that parameters are correctly passed between address spaces

- **Value** parameters
  - Big-endian versus little-endian issues
  - Different sizes of types on different machines
    - E.g., int is 32-bits on x86 platforms and 64-bits on Itanium

- **Reference** parameters (pointers)
  - Pointers are invalid, so entire data structure must be sent
  - What happens if client process updates the structure being pointed to?
  - What should you do with an IN-OUT parameter?

- Thus, a need for *standard* data types and structures ("wire format")
Interface Definition Language (IDL)

- One way for client and server stubs to agree upon parameter passing is to employ a higher-level definition of the procedure’s interface

- Definition in a separate language: Interface Definition Language (IDL)
  - Restricted set of data types
  - Encoding of these data types into messages is standardized
    - call-by-value is straightforward
    - call-by-reference implemented using copy of structure/restore

- Example

```c
struct DateTime {
    long date;
    long time;
};

DateTime getDateTime( void );
```
Translating IDL to Wire Format

Two options

- **Implicit typing**
  - Both the sender and receiver know in advance the type and ordering of data (interface fully defines encoding)
  - E.g., XDR (eXternal Data Representation), NDR (Network Data Repr.)
  - Specifies what byte order is used, what the basic types are, how they are transferred on the wire …
    - E.g., string type is transferred as an int (length) followed by the ASCII bytes

- **Explicit typing**
  - Encoding includes two things
    - a specification of the type and its encoding, and
    - the value in that encoding
  - E.g., ASN.1 (Abstract Syntax Notation 1), BER (Basic Encoding Rules)
Issue 2: RPC Binding

- **Static**
  - RPC server must be running at a **well-known** port number
  - Interaction between clients and servers as in the sockets API

- **Dynamic**
  - Use an intermediate program called a **nameserver**
    - Nameserver must be running at a well-known port
  - Permits binding of server program to port number to be deferred
  - **Server:**
    - RPC server **registers** with nameserver
    - Nameserver allocates a port, and associates it with the server
      - Server listens to a socket bound to this port
  - **Client:**
    - RPC client **looks up** the server by contacting the nameserver
    - Nameserver returns port where server is listening
    - Client sends request to specified port
Dynamic Binding Illustrated

Client

- Look up “S”
- “Bind” to S
- Invoke S (via client stub)

Nameserver

- Allocate port P, Associate P with S
- Respond with P

Server

- Register service S

“export”
Issue 3: Dealing with Failures (State Management)

- Client cannot locate the server
- Lost request

- Server crashes
  - Problem: can crash after processing of request, or before
  - Solutions:
    - at least once - retry until a reply is received
      - requires idempotence (server must generate same reply)
    - at most once - return immediately, client “rebinds” to new server ID

- Client crashes and restarts
  - Problem: computation finished, but client crashed before return (orphan)
  - Solutions:
    - RPC at the client gives a new “incarnation ID” to the client
    - Client has to “rebind” to the service
    - Server uses “client id” to distinguish this instance from the previous one
SunRPC

• Most common implementation of RPC and built into most UNIX OSes
  – Used for Network File System (NFS)

• XDR is used for data description and encoding
  – More about this in the next lecture

• A compiler, rpcgen, translates SunRPC IDL to C, automatically generating
  – Client and server stubs
  – Client and server sample code
  – Header files containing XDR data structure declarations

• A daemon program, portmapper, that provides nameserver functionality
  – Port # 111
Example: A SunRPC Program

![Diagram showing the process of generating client and server programs for a SunRPC application. The process involves the following steps:

1. `date.x`: SunRPC IDL describing parameters, procedure interfaces
2. `rpcgen`: A tool used to generate client and server programs from the IDL.
3. `date_clnt.c` and `date_svc.c`: Generated client and server programs.
4. `date_h`: Header file.
5. `rdate.c` and `date_proc.c`: Additional source code files.

The diagram illustrates the relationship between these components, showing how the IDL is transformed into the necessary files for both client and server programs.]
eXternal Data Representation (XDR)

- A standard for the description and encoding of data
  - Corresponds to the Presentation layer of the ISO protocol stack
- Representation of all items requires a multiple of 4 bytes of data
  - Padding of 0-3 bytes to ensure this condition

- XDR data types
  - integer, unsigned integer, enumeration, boolean, 64-bit signed and unsigned integers, single- and double-precision floating point, fixed-length opaque data
    - Encoding contains only data, no additional information
    - Big-endian byte order
  - Variable-length opaque data, string
    - 4-byte length, followed by the bytes making up the data
XDR (cont’d)

- Fixed-length arrays

- Variable-length arrays

- Structures
Example of XDR Encoding

```c
struct {
    int a;
    int b[4];
} sA;
```

What would the XDR encoding of an instance of `sB` look like, where the field `c` has been allocated to hold 2 elements of type `sA`, and the field `d` holds the string “hi”?

```
struct {
    sA c<>; /* var-length array */
    string d;
} sB;
```

Encoding of `sB`:
```
sA 00 00 00 02 hi
```

Encoding of `c<>`:
```
00 00 00 02 sA sA
```

Encoding of `sA`:
```
a b-0 b-1 b-2 b-3
```
DCE RPC

Distributed Computing Environment RPC

- Open Group standard (also standardized CORBA)
  - DCE: RPC + security, namespace, and network time services
  - Most implementations provide rpcgen-like stub compiler
- Underlying model for Microsoft’s COM implementation

- Data description and encoding: Network Data Representation (NDR)
  - Key difference from XDR is the “receiver-makes-right” model
    - Sender encodes data in format most suited to own architecture
      - Supplies information about encoding in an architecture tag
    - Receiver uses information about encoding to interpret stream
    - Benefit: No translation on homogeneous architectures (LANs)

<table>
<thead>
<tr>
<th>IntRep</th>
<th>CharRep</th>
<th>FloatRep</th>
<th>Extension1</th>
<th>Extension2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: big-endian</td>
<td>0: ASCII</td>
<td>0: IEEE</td>
<td>2: Cray</td>
<td></td>
</tr>
<tr>
<td>1: little-endian</td>
<td>1: EBCDIC</td>
<td>1: VAX</td>
<td>3: IBM</td>
<td></td>
</tr>
</tbody>
</table>

9/17/2003