Assignments

- Progr. assignment 2: due Friday
- HW assignment 1: grades are in dropbox (avg: 76.7%)
- HW assignment 2: due 2/25

Asymmetric Data Conversion

- Programmer could embed knowledge of computer’s architecture into message, one side has to make conversion.
- However, separate version of program needed for each pair of architectures where program used.
- N computers: each uses different data representation, \((N^2 - N)/2\) versions of any client-server program needed.
- Called the ‘n-squared conversion problem’.
- Similar: add new architecture, \(N\) new versions of each client-server pair needed before the new computer can be used with the other computers.

Network Standard Byte Order

- We discussed host/network byte orders already.
- Convert a computer’s native byte order to a network STANDARD byte order.
- This is ‘symmetric data conversion’.
- Each end agrees to convert between individual representation and standard representation.
- Only one version of protocol software is needed.
- Also known as ‘external data representation’.
- Advantage: flexibility (I don’t have to understand the architecture of your computer...), simplifies programming, reduces errors, increases interoperability among programs, makes management and debugging easier...
- Disadvantage: computational overhead (we have to convert even if we have the same architecture; both ends have to convert instead of one end); data stream may increase (adding information).

De Facto Standard

- XDR
- Sun’s eXternal Data Representation
- Specifies data formats for most data types that clients/servers exchange (e.g., 32-bit integers are represented as ‘big endian’).

XDR

- Process A
  - XDR Encode/Decode
  - Transport
  - Process A
  - XDR Encode/Decode
  - Transport
XDR Terminology

- Converting from local representation to XDR representation is called **Encoding**.
- Converting from XDR representation to local representation is called **Decoding**.

![XDR Diagram](image)

XDR Data Types

- **int** 32 bits integer
- **unsigned int** 32 bits unsigned integer
- **bool** 32 bits boolean value (false/true)
- **enum** arb. enumeration type
- **hyper** 64 bits 64 bit integer
- **unsigned hyper** 64 bits 64 bit unsigned integer
- **float** 32 bits single prec. FP
- **double** 64 bits double prec. FP
- **opaque** arb. unconverted data
- **string** arb. string
- **fixed array** arb. fixed-size array
- **counted array** arb. array with fixed upper limit
- **structure** arb. like C's struct
- **void** 0 no data present (structure)

![XDR Data Types Table](image)

Implicit Types

- XDR encodes only data items, not information about their types:
  - 32 bit integer is stored big endian, result occupies exactly 32 bits
  - encoding DOES NOT contain additional information to identify it as an integer or to specify length
- Clients and servers using XDR must agree on exact format of messages they will exchange. A program cannot interpret an XDR-encoded message unless it knows the exact format and the types of all data fields.

![Implicit Types Diagram](image)

XDR Library Routines

- Most implementations of XDR use the “buffer paradigm”:
  - program has to allocate a buffer large enough to hold the external representation of a message and to add items (‘fields’) one at a time
  - `xdrmem_create`: allocate a buffer in memory and inform XDR about intention to compose an external representation in it. Memory is initialized to be empty (internal pointer points to beginning of ‘stream’), the call returns a pointer to the memory (or ‘stream’)

![XDR Library Routines 1](image)

```c
#include <rpc/xdr.h>
define BUFSIZE 4000

XDR *xdrs; /* pointer to XDR 'stream' */
char buf[BUFSIZE]; /* memory to hold XDR data */
xdrmem_create(xdrs, buf, BUFSIZE, XDR_ENCODE);

// you can call individual XDR conversion routines now
// each call encodes one data object and appends the
// encoded information on the end of the stream (next
// available location in the buffer) and updates the stream
// pointer
```

![XDR Library Routines 2](image)

```
xdr_int: converts 32-bit integer from native representation to XDR representation and appends it to an XDR stream.
```

```c
int i; /* integer in native representation */
...
i = 260;
xdr_int(xdrs, &i); /* conversion */
```
XDR Library Routines

- After encoding all items and placing them into the stream, you can send the stream.
- Receiving application:
  - `xdrmem_create`: create a buffer that will hold an XDR stream
  - place incoming message into buffer
  - receiver specifies XDR_DECODE when calling `xdrmem_create`
  - call conversion routines: XDR will convert items to native mode

XDR Streams, I/O, TCP

- Review: create buffer, indicate conversion direction, convert, send/use.
- Alternative: XDR can send data across a TCP connection each time a data item is converted to external form:
  - application creates TCP socket
  - call `fdopen` to attach a standard I/O stream to the socket
  - call `xdrstdio_create` instead of `xdrmem_create`
  - connect XDR stream to existing I/O descriptor
  - no read/write calls needed

XDR Library Routines

- `xdr_bool`
- `xdr_char`
- `xdr_double`
- `xdr_float`
- `xdr_int`
- `xdr_long`
- `xdr_opaque`
- `xdr_short`
- `xdr_string`
- `xdr_u_short`
- `xdr_u_int`
- ...

XDR Streams, I/O, TCP

```c
#include <stdio.h>
#include <rpc/xdr.h>

void xdrstdio_create (xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;

xdrstdio_create(xdrs, fd, XDR_ENCODE);
```
XDR Streams, I/O, TCP

```c
#include <stdio.h>

FILE *fd;
fd = fdopen (sd, "w");
```

- `sd` is open descriptor (socket)
- Second parameter is mode (write, read, ...)

Records, Record Boundaries, Datagram I/O

- **UDP**: record-oriented interface
- **xdrrec_create**:
  - Parameter `inproc`: input procedure
  - Parameter `outproc`: output procedure
- Converting to external form: each conversion routine checks the buffer; if buffer becomes full, the routine calls `outproc` to send buffer content and make space for new data.
- Converting to internal form: routine checks buffer if it contains data; if buffer is empty, the routine calls `inproc` to obtain more data

Records, Record Boundaries, Datagram I/O

- `outproc`: write buffer to network in a single datagram.
- `inproc`: read from network

Back to RPC

- Uses XDR language.
- Example: use XDR to define a message type field in the RPC message header:

  ```c
enum msg_type {
    CALL = 0;
    REPLY = 1;
  };
```

- Distinguish between request and response

RPC + XDR

- Message:

  ```c
  struct rpc_msg {
    unsigned int msgid; /* match reply to call */
    union switch (msg_type mesgt) {
      case CALL:
        call_body cbody;
      case REPLY:
        reply_body rbody;
    } body;
  };
  ```

RPC + XDR

- Example for `call_body`:

  ```c
  struct call_body { /* format of RPC CALL */
    unsigned int rpcvers;
    unsigned int rprog;
    opaque_auth cred;
    opaque_auth verf;
    /*ARGS*/
  };
  ```
Authentication

- Several possible forms of authentication, e.g., DES (Data Encryption Standard).

  ```c
  enum auth_type {
      AUTH_NULL = 0; /* no authentication */
      AUTH_UNIX = 1; /* machine name authentic */
      AUTH_SHORT = 2; /* short form auth. */
      AUTH_DES = 3;
  };
  ```

RPC+XDR

- RPC offers a lot of help:
  - XDR library for converting data items
  - XDR library for forming complex data aggregates for RPC messages
  - RPC run-time library for calling remote procedure, registering a service with the port mapper, dispatch incoming calls to procedures
  - program generator that produces many of the C source files needed

More Calls...

```c
int callrpc(host, prog, progver, procnum, inproc, in, outproc, out);
```
- host: string containing remote machine name
- prog/progver/procnum: ids
- inproc: address of a local procedure that can be called to marshal arguments into an RPC message
- in: address of arguments for remote procedure
- outproc: address of a local procedure that can be called to decode the results
- out: address in memory where the results should go

More Calls

```c
handle = clnt_create(host, prog, vers, proto);
```
- integer identifier to send RPC messages
- remote hostname, program, version, protocol

```c
authunix_create(host, uid, gid, len, aup_ids);
```
- authentication handle: remote host, user login, group id, set of groups

Example

- Chapter 23:
  - dictionary database
  - source code available at: http://www.cs.purdue.edu/homes/dec/netbooks.html (look for our textbook, there is a link to the code, the dictionary code is in the RPC.ex subdirectory)
Java RMI

- RMI = Remote Method Invocation.
- Allows Java programs to invoke methods of remote objects.
- Only between Java programs.
- Several versions (JDK-1.1, JDK-1.2)

Remote Method Invocation

Interfaces (Transparency)

- To client, remote object looks exactly like a local object (except that you must bind to it first).
- Using interfaces:
  - you write interface for remote object
  - you write implementation for remote object
  - RMI creates stub class (implementing the remote object interface)
  - client accesses stub exactly same way it would access a local copy of the remote object

RMI Registry

- RMI needs a port mapper too:
  - servers can register contact address information
  - clients can locate servers
- Called RMI registry
- You must start it yourself (unlike RPC):
  - needs to be started on every machine that hosts server objects
  - runs on port 1099 by default (but you can use `rmiregistry <port_nb>`)  
- Programs can access the registry thanks to java.rmi.Naming class.

Example

- Simple program:
  - write interface for remote object: Remote.java
  - implementation of object: RemoteImpl.java
  - server to run object: RemoteServer.java
  - client to access object: Client.java
- RMI compiler ‘rmic’ generates:
  - client stub: RemoteImpl_Stub.class (already compiled)
  - server skeleton: RemoteImpl_Skel.class (already compiled)
Example

Step 1: write interface Calculator.java
import java.rmi.Remote;
public interface Calculator extends Remote {
    public long add(long a, long b) throws RemoteException;
    public long sub(long a, long b) throws RemoteException;
    public long mul(long a, long b) throws RemoteException;
    public long div(long a, long b) throws RemoteException;
}

Few rules:
- interface must extend java.rmi.Remote interface
- methods must throw java.rmi.RemoteException exception

Compile:
- $ javac Calculator.java

Example

Step 2: write remote object CalculatorImpl.java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
public class CalculatorImpl extends UnicastRemoteObject implements Calculator {
    // Implementations must have an explicit constructor
    public CalculatorImpl() throws RemoteException {
        super();
    }
    public long add(long a, long b) throws RemoteException {
        return a + b;
    }
    public long sub(long a, long b) throws RemoteException {
        return a - b;
    }
    public long mul(long a, long b) throws RemoteException {
        return a * b;
    }
    public long div(long a, long b) throws RemoteException {
        return a / b;
    }
}

Example

Step 3: generate stub and skeleton
RMI compiler:
- $ rmic CalculatorImpl
Generates CalculatorImpl_Stub.class and CalculatorImpl_Skel.class files.
- Already compiled.

Example

Step 4: write server CalculatorServer.java
import java.rmi.Naming;
public class CalculatorServer {
    public CalculatorServer() {
        try {
            Calculator c = new CalculatorImpl();
            Naming.rebind("rmi://localhost:1099/CalculatorService", c);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }
    public static void main(String args[]) {
        new CalculatorServer();
    }
}

Example

Step 5: write server CalculatorServer.java
- Server program creates CalculatorImpl object.
- Registers object to local RMI registry (rebind()):
  - rebind(String name, Remote obj) associates a name to an object
  - names are in the form of a URL: rmi://<host_name>:<port>/<service_name>
- Server waits for incoming requests
- $ javac CalculatorServer.java
Example

Step 5: write CalculatorClient.java

```java
import java.net.MalformedURLException;
public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Calculator c = (Calculator) Naming.lookup(
                "rmi://wizard.cse.nd.edu/CalculatorService")
                System.out.println(c.add(4,5));
                System.out.println(c.sub(4,3));
        } catch (Exception e) {
            System.out.println("Received Exception:");
            System.out.println(e);
        }
    }
}
```

Before invoking the server, the client must ‘lookup’ the registry:
- must provide the URL for remote service
- gets back a stub which has exactly the same interface as the server
- can use it as a local object: `long x = c.add(4,5);`

Compile: `javac CalculatorClient.java`

Example

Step 6: test it!

Start the RMI registry: `rmiregistry`
- registry must have access to your classes
- either start the registry in the same directory as the classes or make sure directory is listed in `$CLASSPATH` variable

Start server: `java CalculatorServer`

Start client: `java CalculatorClient`
```
9
1
```

Using RMI in a Distributed Context

First, test your program on a single host.
To use it on 2 machines:
- server and `rmiregistry` need the following files:
  - `Calculator.class` (server object interface)
  - `CalculatorImpl.class` (server object implementation)
  - `CalculatorImpl_Stub.class` (stub)
  - `CalculatorServer.class` (server program)
- client needs:
  - `Calculator.class` (server object interface)
  - `CalculatorImpl_Stub.class` (stub)
  - `CalculatorClient.class` (client program)
- nobody needs the skeleton file `CalculatorImpl_Skel.class`
- generated only for compatibility with JDK-1.1

Method Parameters Transfer

- Base types (`int`, `float`, `char`) are transferred directly.
- Remote objects (inheriting from `java.rmi.Remote`) are not transferred:
  - instead, a distributed reference to object is shipped
  - any invocation to this object will result in a RMI request
- Non-remote objects are serialized and shipped:
  - the object itself plus every other object that it refers to (recursively)
  - remote invocations pass objects by value (local by reference)
  - very easy to transfer huge quantities of data without noticing
    (you have database in memory and you transfer an object which contains a reference to database)