V22.0480-004
Web Services Architecture and Programming

Lecture 4
Sockets API (cont’d)
Remote Procedure Calls

Sources:

*UNIX Network Programming, Volume 1* (W. Richard Stevens)
Internet RFCs, …
Announcements

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

• Two TA’s for the course
  – Congchun He (congchun@cs.nyu.edu)
    • Wednesdays 2:00pm – 4:00pm, 715 Broadway, 7th Floor
  – Ilya Rosenberg (ilya@cs.nyu.edu)
    • Mondays 4:00pm – 6:00pm, 715 Broadway, 12th Floor
(Review) **Setting Up and Using Sockets – TCP**

**Client**

```java
Client

\[\text{t = new Socket(...);}\]
```

Try connecting with server: OS implicitly associates a port, sends SYN packet

```java
\text{t.Connect([Server, SPort]);}
```

Client blocks

**Server**

```java
\[\text{s = new Socket(...);}\]
```

Associate a well-known port number with the server-end of application

```java
\text{s.Bind([Server, SPort]);}
```

Places the socket in a listening state: OS starts queuing SYN packets

```java
\text{s.Listen(#connections);}
```

Waits for handshake, and returns a new socket (bound to a different port)

```java
\text{u = s.Accept();}
```

Server blocks
(Review) Setting Up and **Using** Sockets – TCP

**Client**
- \texttt{t.Send(...)};
- \texttt{t.Receive(...)};

**Server**
- \texttt{s}: listening socket
- \texttt{u}: accepted socket
- \texttt{u.Receive(...)};
- \texttt{v = s.Accept(...)};
- \texttt{u.Send(...)};

Whether or not Send blocks is determined by buffer associated with socket

Receive blocks till sufficient data has been read (or connection closed)

Accept a new connection. Again, returns a new socket (bound to a different port)
Example: A Simple Server Program (StringServer)

[  
  Code walk-through using Remote Desktop Connection to netserver1.pdsg.cs.nyu.edu  
  StringServer code available as part of Lab1 helper files.  
]

9/15/2003
Programming With The Sockets API

**Byte streams**, no boundaries preserved
- Send’s and Receive’s can line up arbitrarily
- Therefore, need a **convention** about data format
  - For our StringServer app: `<length> <sequence of bytes>`
  - For HTTP packets: server parses client requests
    - HTTP standard defines the format of these requests
- Your networking programs need to work in heterogeneous environments
  - **Byte-order** (Endian-ness) matters: network byte order is big-endian
  - HostToNetworkOrder, NetworkToHostOrder functions
- Errors can arise because of a number of reasons
  - Connect request to a socket that is not being listened to, early close, disconnected hosts, …
  - In C#, .NET, all of these delivered to the application as exceptions
Sockets API: Building Concurrent Servers

Handling each client connection request as its own thread frees up the main server thread so it can accept additional connections.

New connections are blocked

More scalable alternative: Event-driven programming
Sockets API: Advanced Functions

• Consider a server that is listening to two sockets: s1 and s2
  – How can it accept connections/receive data that appear on either …
  – … without knowing beforehand which of them will have activity
  – **Problem**: Accept/Receive are blocking calls

```cpp
bool Poll( int usecs, SelectMode mode )
- **mode**: SelectRead, SelectWrite, SelectError
  – Query state of a socket: returns immediately without blocking
    • Returns true if socket is ready for reading, writing, or an error is pending

void Select( IList checkRead, IList checkWrite, IList checkError, int usecs )
- Poll functionality extended to multiple sockets
  - **check{Read,Write,Error}** are INOUT parameters
    • IN: socket(s) whose state needs to be checked
    • OUT: socket(s) that are in fact ready```
Sockets API: Advanced Functions (cont’d)

- Functions to read/set various socket options
  - A large number of these controlling socket characteristics (buffer sizes), IP and TCP protocol parameters (such as TTL, retransmit times, …)
  - See `System.Net.Sockets.SocketOptionName` enumeration for details

- Winsock 2 supports asynchronous and overlapped calls
  - Permits network and computation operations to be overlapped
    - Allows event-driven programming
  - .NET framework library: Additional methods of the Socket class
    ```
    Begin{Connect, Listen, Accept, Send, SendTo, Receive, ReceiveFrom}
    End{ ... }
    ```
Remote Procedure Calls (RPC)

- Writing distributed applications using the Sockets API is complicated
  - Program needs to explicitly send/receive messages
  - Violates programming transparency
    - Local application components
    - Remote application components

RPC: Remote Procedure Call

- Original goal: Provide complete programming transparency
  - Interaction with “procedures” on remote hosts as if they were local
  - However, needed to deal with several issues
    - Different memory spaces, Parameter passing, Binding, Failures

- Now: A higher-level abstraction for distributed programming
  - Provides near-transparency
RPC History

• Original idea described in a 1983 paper by A. Birell and B. J. Nelson

• Several implementations
  – Xerox
  – SunRPC: widely used (NFS)
  – DEC
  – DCE (OpenGroup): basis for Microsoft’s implementation of RPC in COM
  – ...
  – XML-RPC
    • same underlying ideas, but leverages web standards
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: Local Procedure Calls

- Steps in a local procedure call
  - Caller pushes parameters, return address on a stack
  - Control transferred to procedure
  - Procedure returns values in registers, removes return address and passes control back
    - Call-by-value
    - Call-by-reference
  - Caller cleans up stack frame

- How can we emulate these in a network setting?
  - Parameter passing across address spaces
    - Pointers are not valid across machines
  - Binding
    - Static: compile/link time (as in the local procedure case)
    - Dynamic: using an intermediary service, requires registration and lookup
  - Dealing with failures
    - Client and server crashes
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

SunRPC IDL describing parameters, procedure interfaces

date.x

rpcgen

date_clnt.c date.h
date_svc.c
date_svc.h
date_proc.c

rdate.c

RPC runtime

Client program

Server program
/*
   * dateproc.c - remote procedures; called by server stub.
   */

#include <rpc/rpc.h>     /* standard RPC include file */
#include "date.h"        /* this file is generated by rpcgen */

/*
 * Return the binary date and time.
 */

long *
bin_date_1()
{
    static long     timeval;        /* must be static */
    long            time();         /* Unix function */

    timeval = time((long *) 0);

    return(&timeval);
}

/*
 * Convert a binary time and return a human readable string.
 */

char **
str_date_1(bintime)
long    *bintime;
{
    static char     *ptr;           /* must be static */
    char            *ctime();       /* Unix function */

    ptr = ctime(bintime);           /* convert to local time */

    return(&ptr);                    /* return the address of pointer */
}
/*
 * date.x - Specification of remote date and time service.
 */

/*
 * Define 2 procedures:
 *   bin_date_1() returns the binary time and date (no arguments).
 *   str_date_1() takes a binary time and returns a human-readable string.
 */

program DATE_PROG {
  version DATE_VERS {
    long     BIN_DATE(void) = 1;  /* procedure number = 1 */
    string   STR_DATE(long) = 2; /* procedure number = 2 */
  } = 1;                       /* version number = 1 */
  } = 0x31234567;              /* program number = 0x31234567 */
                    /* 0x20000000 - 0xffffffff for users*/
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _DATE_H_RPCGEN
#define _DATE_H_RPCGEN

#include <rpc/rpc.h>

#define DATE_PROG ((unsigned long)(0x31234567))
#define DATE_VERS ((unsigned long)(1))
#define BIN_DATE ((unsigned long)(1))
extern long * bin_date_1();
#define STR_DATE ((unsigned long)(2))
extern char ** str_date_1();
extern int date_prog_1_freeresult();

#endif /* !_DATE_H_RPCGEN */
/*
 * rdate.c - client program for remote date service.
 */

#include <stdio.h>
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */

main(argc, argv)
int argc;
char *argv[];
{
    CLIENT *cl; /* RPC handle */
    char *server;
    long *lresult; /* return value from bin_date_1() */
    char **sresult; /* return value from str_date_1() */

    if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]);
        exit(1);
    }

    server = argv[1];

    /*
     * Create the client "handle."
     */

    if ( (cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
        /*
         * Couldn’t establish connection with server.
         */

        clnt_pcreateerror(server);
        exit(2);
    }

    /*
     * First call the remote procedure "bin_date".
     */

    if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
        clnt_perror(cl, server);
        exit(3);
    }

    printf("time on host %s = %ld\n", server, *lresult);

    /*
     * Now call the remote procedure "str_date".
     */

    if ( (sresult = str_date_1(lresult, cl)) == NULL) {
        clnt_perror(cl, server);
        exit(4);
    }

    printf("time on host %s = %s", server, *sresult);

    clnt_destroy(cl); /* done with the handle */
    exit(0);
}
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "date.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

long *
bin_date_1(argp, clnt)
    void *argp;
    CLIENT *clnt;
{
    static long clnt_res;
    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, BIN_DATE,
        (xdrproc_t) xdr_void, (caddr_t) argp,
        (xdrproc_t) xdr_long, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

char **
str_date_1(argp, clnt)
    long *argp;
    CLIENT *clnt;
{
    static char *clnt_res;
    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, STR_DATE,
        (xdrproc_t) xdr_long, (caddr_t) argp,
        (xdrproc_t) xdr_wrapstring, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "date.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#include <signal.h>
#include <sys/types.h>
#include <memory.h>
#include <stropts.h>
#include <netconfig.h>
#include <sys/resource.h> /* rlimit */
#include <syslog.h>

#ifdef DEBUG
#define RPC_SVC_FG
#endif

#define _RPCSVC_CLOSEDOWN 120
static int _rpcpmstart;        /* Started by a port monitor ? */

/* States a server can be in wrt request */
#define _IDLE 0
#define _SERVED 1

static int _rpcsvccircuit = _IDLE;        /* Set when a request is serviced */
static int _rpcsvccount = 0;            /* Number of requests being serviced */

static
void _msgout(msg)
    char *msg;
{
    #ifdef RPC_SVC_FG
    if (_rpcpmstart)
        syslog(LOG_ERR, msg);
    else
        (void) fprintf(stderr, "%s\n", msg);
    #else
        syslog(LOG_ERR, msg);
    #endif
}

static void
closedown(sig)
    int sig;
{
    if (_rpcsvccircuit == _IDLE && _rpcsvccount == 0) {
        extern fd_set svc_fdset;
        static int size;
        int i, openfd;
        struct t_info tinfo;

        if (!t_getinfo(0, &tinfo) && (tinfo.servtype == T_CLTS))
            exit(0);
        if (size == 0) {
            struct rlimit rl;

            rl.rlim_max = 0;
            getrlimit(RLIMIT_NOFILE, &rl);
            if ((size = rl.rlim_max) == 0) {
                return;
            }
        }
    }
for (i = 0, openfd = 0; i < size && openfd < 2; i++)
    if (FD_ISSET(i, &svc_fdset))
        openfd++;
if (openfd <= 1)
    exit(0);
else
    _rpcsvcstate = _IDLE;

(void) signal(SIGALRM, (void (*)(void *)) closedown);
(void) alarm(_RPCSVC_CLOSEDOWN/2);

static void
dateprog_1(rqstp, transp)
    struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    union {
        long str_date_1_arg;
    } argument;
    char *result;
    bool_t (*xdr_argument)(), (*xdr_result)();
    char *(*local)();

    _rpcsvccount++;
    switch (rqstp->rq_proc) {
    case NULLPROC:
        (void) svc_sendreply(transp, xdr_void,
            (char *)NULL);
        _rpcsvccount--;
        _rpcsvcstate = _SERVED;
        return;
    case BIN_DATE:
        xdr_argument = xdr_void;
        xdr_result = xdr_long;
        local = (char *(*)(* )()) bin_date_1;
        break;
    case STR_DATE:
        xdr_argument = xdr_long;
        xdr_result = xdr_wrapstring;
        local = (char *(*)(* )()) str_date_1;
        break;
    default:
        svcerr_noproc(transp);
        _rpcsvccount--;
        _rpcsvcstate = _SERVED;
        return;
    }
    (void) memset((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs(transp, xdr_argument, &argument)) {
        svcerr_decode(transp);
        _rpcsvccount--;
        _rpcsvcstate = _SERVED;
        return;
    }
    result = (*local)(&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
        svcerr_systemerr(transp);
    }
}
if (!svc_freeargs(transp, xdr_argument, &argument)) {
    _msgout("unable to free arguments");
    exit(1);
}
_rpcsvccount--;
_rpcsvcstate = _SERVED;
return;
}

main()
{
    pid_t pid;
    int i;
    char mname[FMNAMESZ + 1];

    (void) sigset(SIGPIPE, SIG_IGN);

    if (!ioctl(0, I_LOOK, mname) &&
        (!strcmp(mname, "sockmod") || !strcmp(mname, "timod"))) {
        char *netid;
        struct netconfig *nconf = NULL;
        SVCXPRT *transp;
        int pmclose;

        _rpcpmstart = 1;
        openlog("date", LOG_PID, LOG_DAEMON);

        if ((netid = getenv("NLSPROVIDER")) == NULL) {
            /* started from inetd */
            pmclose = 1;
        } else {
            if ((nconf = getnetconfig(netid)) == NULL)
                _msgout("cannot get transport info");

            pmclose = (t_getstate(0) != T_DATAXFER);
        }

        if (strcmp(mname, "sockmod") == 0) {
            if (ioctl(0, I_POP, 0) || ioctl(0, I_PUSH, "timod")) {
                _msgout("could not get the right module");
                exit(1);
            }
        }

        if ((transp = svc_tli_create(0, nconf, NULL, 0, 0)) == NULL) {
            _msgout("cannot create server handle");
            exit(1);
        }

        if (nconf)
            freenetconfig(nconf);

        if (!svc_reg(transp, DATE_PROG, DATE_VERS, date_prog_1, 0)) {
            _msgout("unable to register (DATE_PROG, DATE_VERS)."));
            exit(1);
        }

        if (pmclose) {
            (void) signal(SIGALRM, (void(*)()) closedown);
            (void) alarm(_RPCSVC_CLOSEDOWN/2);
        }

        svc_run();
        exit(1);
        /* NOTREACHED */
    } else {
#elifdef RPC_SVC_FG
    int size;
    struct rlimit rl;
    pid = fork();
if (pid < 0) {
    perror("cannot fork");
    exit(1);
}
if (pid)
    exit(0);
rl.rlim_max = 0;
getrlimit(RLIMIT_NOFILE, &rl);
if ((size = rl.rlim_max) == 0)
    exit(1);
for (i = 0; i < size; i++)
    (void) close(i);
i = open("/dev/console", 2);
(void) dup2(i, 1);
(void) dup2(i, 2);
setsid();
openlog("date", LOG_PID, LOG_DAEMON);
#endif
}
if (!svc_create(date_prog_1, DATE_PROG, DATE_VERS, "netpath")) {
    _msgout("unable to create (DATE_PROG, DATE_VERS) for netpath.");
    exit(1);
}
svc_run();
_msgout("svc_run returned");
exit(1);
/* NOTREACHED */