Distributed Data Retrieval Infrastructure

The project is aiming to address the problem of distributed data retrieval from remote hosts in a platform transparent manner. Specifically, it was designed to retrieve the coordinate information produced by a camera attached to a remote host. The main goal of the project is to let someone developing a C++ program, not necessarily familiar with network programming, easily obtain information from a remote host. Another goal of the project is to make the infrastructure as platform independent as possible, which means a host to which the camera is attached can be running either Linux or Windows; also the developed client program which would use the developed library could be running in either of these platforms. Finally, yet another goal of the project is to provide an object interface in a typical C++ manner.

The main development platform of the project is Linux. The portability is achieved through compilation of the same source tree on Windows platform using either Cygwin[1] or MinGW[2] environments. Cygwin is a Unix-like environment and command-line interface for Windows. Cygwin provides native integration of Windows-based applications, data, and other system resources with applications, software tools, and data of the Unix-like environment. Similarly, MinGW is a port of the GNU Compiler Collection (GCC), and GNU Binutils, for use in the development of native Windows applications. Offered in easily installed binary package format, for native deployment on Windows, or user-built from source, for cross-hosted use on Unix or Linux, the suite exploits Microsoft's standard system DLLs to provide the C-Runtime and Windows API. While the portability could be achieved through having two different versions of the program utilizing BSD Sockets API[3] or Windows Socket API[4], either completely different source trees or sources having #ifdef's all over, it was not chosen since maintaining such a source tree would be an extra burden. Both Cygwin and MinGW provide necessary environment for easy compilation of Linux programs on Windows platform.

The project consists of two parts -- a library which basically implements a camera object, containing of a C++ header and the implementation of the camera object, and a server. A typical usage is running a server on the machine to which the camera is physically attached. On startup, the

server would run and collect the data from camera and make it available on TCP port 3333; the client functionality is utilized by constructing a camera object which requires an IP address of the machine on which server process is running; once constructed, issuing read() calls on this object would produce a pair of coordinates to be consumed by the client program. It should be noted that camera object construction as well as read() calls should always be within try-catch blocks since these involve networking calls that may return an error. Below is the source code for a simple driver:

```cpp
#include <iostream>
#include <stdexcept>
#include "camera.hpp"          // camera header

using namespace std;

int main(int argc, char *argv[])
{
    if (argc != 2) {
        cerr << "Usage: driver IP" << endl;
        return 1;
    }
    try {   // try-catch block, constructor may fail if the remote server is not up
        double x, y;
        camera r_camera(argv[1]);  // construct a camera object; if argv[1] is not a
valid IP address it will throw an exception
        while (r_camera.read(x, y))  // keep reading coordinates
            cout << x << ", " << y << endl;  // and printing them out
    } catch (runtime_error e) {
        cout << e.what() << endl;
    }

    return 0;
}
```

Below a detailed workings of both server and client will be provided for those wishing to extend

or modify the functionality. We will start with the server. Since the server is a stand-alone program that does not necessarily be open to the end-users of the library, it was written in ANSI C. The main reference for the development is the seminal work of Richard Stevens[5]. It's a definitive reference on the sockets API on Unix environments that does a thorough walk through of sockets programming gotchas.

The server is concurrent, which means it can handle many clients simultaneously. It achieves the concurrency by creating a new process for handling every connected client the details of which will follow. Its code resides in a file aptly named server.c; besides the main function it contains five functions which will be discussed next. We will start with the main function:

```c
int main(int argc, char *argv[])
{
    int listenfd, connfd;
    pid_t childpid;
    socklen_t clilen;
    struct sockaddr_in cliaddr, servaddr;
    void sig_chld(int);

    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        errx(1, NULL);

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SRV_PORT);

    if (bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
        errx(1, NULL);

    if (listen(listenfd, 10) < 0)
        errx(1, NULL);
```

```
    if (signal(SIGCHLD, sig_child) == SIG_ERR)

        errx(1, NULL);


    for ( ; ; ) {

        clilen = sizeof(cliaddr);

        if ((connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen)) < 0) {

            if (errno == EINTR)

                continue;

            else

                errx(1, NULL);

        }


        if ((childpid = fork()) == 0) {    /* child process */

            if (close(listenfd) == -1)

                errx(1, NULL);

            process(connfd);    /* process the request */

            exit(0);

        } else if (childpid == -1)

            errx(1, NULL);

        if (close(connfd) == -1)

            errx(1, NULL);

    }

}
```

The server starts by creating a socket and binding it to port SRV_PORT, which is a macro defined to have a value of 3333 by default. These two actions are performed by socket() and bind() calls from the sockets API. Next the server starts listening on the socket for incoming connections, performed by the listen() call. It is followed by a signal() call, which registers a handler -- which currently does nothing of importance, it just prints the process ID of the dead process -- for the signal SIGCHLD, which is a signal that will be delivered to the server whenever one of its children processes dies; as mentioned previously the server achieves concurrency by creating multiple processes. While it could be argued

that concurrency should have been achieved through the use of some threads API like POSIX Threads[6] instead of process creation, since it has a high overhead due to context switches and process forking is slow and resource inefficient, the counter argument is that a typical usage of the system will not have thousands of clients connected to it and in this case the complexity added by a threads API is not worth the trouble. Once the signal handler is registered, the process goes to an infinite loop where it keeps accepting new connections through the accept() call and forking new process for each successful connection. It is worth noting that the fork() call returns twice, once in the parent process -- where it returns the process ID of the created process -- and once in the child process -- where it returns 0. For the child another call is made to process() function which is defined by us and will be described next; for the parent process the connection descriptor is closed since it is of no use to parent, whose job is to accept connections and fork child processes. This summarizes the workings of the main function. Next we will look at the process function:

```
static void process(int sockfd)
{
    ssize_t n;
    char line[MAXLINE];

    for (;;) {
        if ((n == readline(sockfd, line, MAXLINE)) < 0)
            errx(1, "%s", "failed to read from client");
        else if (n == 0 || strncmp("exit\n", line, MAXLINE) == 0)
            return;
        sprintf(line, "%d %d\n", rand(), rand());
        if (writen(sockfd, line, strlen(line)) != strlen(line))
            errx(1, "%s", "failed to write to client");
        sleep(1);
    }
}
```

This is the function called in the child process once a connection is established. Basically this is the only function that will need to be modified for providing data different than the default, which as can

be seen two random numbers. The protocol between the client and server is line based. Once a TCP

connection is established between the server and the client, the server will wait until the client sends an

empty line; once the server reads the empty line it responds with a line consisting of two numbers, in

this case. Of course in real deployment this part of the process function will be replaced with calls to

functions that read data from the camera interface and fill the line and send it to the client. Once the

clients are done processing, they could send a line with "exit\n" string which will cause the child server

process to close the socket and die; note that this is not the main process that listens on port 3333 but

the one that was forked to handle a specific client. The server will also detect whenever the remote

client process is dead, since a TCP FIN packet will be sent to the server which will cause it to read 0

bytes from the socket, which will cause the related server process to die gracefully. The final sleep()

call above is for the demo purposes only, so that one could observer values read from the client when

compiling and running the demo. It should definitely be removed in production code. Next we will look

at the readline() function which is actually trickier to get right than it may seem:

```
static ssize_t readline(int fd, void *vptr, size_t maxlen)
{
    int        n, rc;
    char    c, *ptr;

    ptr = (char *)vptr;
    for (n = 1; n < maxlen; n++) {
        if ( (rc = my_read(fd, &c)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            if (n == 1)
                return 0;
            else
                break;
        } else
            return -1;
```

```
        }

    *ptr = 0;

    return n;

}
```

The function reads characters from the supplied descriptor, which corresponds to the socket between the client and the server, into the supplied buffer one by one so that it can detect line endings and return a complete line with the newline at the end as a C string, i.e. null-terminated. As can be seen it utilizes my_read() function for reading characters one by one. It null-terminates the string and returns the number of characters read as the return value; my_read() function does most of the reading work from the socket:

```
static ssize_t my_read(int fd, char *ptr)
{
    static int    read_cnt = 0;
    static char    *read_ptr;
    static char    read_buf[MAXLINE];

    if (read_cnt <= 0) {
    again:
        if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            if (errno == EINTR)
                goto again;
            return -1;
        } else if (read_cnt == 0)
            return 0;
        read_ptr = read_buf;
    }
    read_cnt--;
    *ptr = *read_ptr++;
    return 1;
}
```

As the source shows, my_read() performs internal buffering; i.e. it does not read characters one by one from the socket. It performs one huge read from the descriptor and keeps returning a single character until the buffer is depleted; once so, it does another huge read. Together with readline() and my_read() an efficient buffering achieved which also gives us ability to detect line endings; my_read() also handles the cases where a read() call could have been interrupted, in which case one should keep retrying a read() call unless the errno value is different than EINTR which may mean a more serious problem. A good treatment of all these subjects can be found in [5]. Another important function is the writen() function again utilized by the process() function:

```
static ssize_t writen(int fd, const void *vptr, size_t n)
{
    size_t        nleft;
    ssize_t        nwritten;
    const char    *ptr;

    ptr = (const char *)vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (errno == EINTR)
                nwritten = 0;
            else
                return -1;
        }
        nleft -= nwritten;
        ptr    += nwritten;
    }
    return(n);
}
```

The reason why the write() system call was not utilized in the process() can be understood by looking at the above code. It's possible and legal for a write() call to be interrupted and errno set to EINTR; as with the read, one should not stop doing write() and return an error upstream but should continue trying to write() unless either the errno is something other than EINTR or the supplied buffer is completely written to the provided descriptor, which is again the socket between client and the server in our case. The final function in the server code is the signal handler, which does not do anything special other than printing the process ID of the dying process. The handler is provided for avoiding zombie processes and could probably utilized for better purposes like reporting the number of bytes written by a specific process, etc.

```
static void sig_child(int signo)
{
    pid_t pid;
    int    stat;

    while ((pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated\n", pid);
    return;
}
```

Next the client code is considered. As mentioned previously the client functionality is implemented as a C++ object. The header file called camera.hpp should be included by applications utilizing the library is described below:

```
class camera {
public:
    camera(const std::string& ip);
    bool read(double& x, double &y);
private:
    std::string remote_ip;
    struct sockaddr_in server_addr;
    int sockfd;
};
```

The interface of the camera object consists of a constructor which takes the IP address as string parameter and a read() function which takes as input two double parameters which will be filled with x and y coordinate values read from the server. As private members, project holds the IP address, the socket address structure and the file descriptor number of the created socket through which it will communicate with the server. Below is the code for the object constructor:

```
camera::camera(const string& ip): remote_ip(ip)
{
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port   = htons(SRV_PORT);


    if (inet_pton(AF_INET, remote_ip.c_str(), &server_addr.sin_addr) <= 0)
        throw runtime_error("not a valid IP address");


    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        throw runtime_error("failed to create a socket");


    if (connect(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0)
        throw runtime_error("failed to connect to remote server");
}
```

The constructor starts by creating a socket address structure and initializing its sin_port member to SRV_PORT which is 3333 by default. It should be noted that if the port number is changed in the server code, the client code should also be changed and recompiled. Next, the constructor validates and converts the IP address given as an ASCII string to its binary form; in case of failure an exception is thrown; hence all calls to camera constructor should be in a try-catch block. Once the IP is converted to binary, a call to socket() is made and a socket descriptor obtained, which is saved in the private member sockfd. Finally, connect() call is made to establish a connection with the server. At this step one could call read() member of the camera object whenever there is a need for reading coordinate data from the remote server; read() call is implemented as follows:

```cpp
bool camera::read(double& x, double& y)
{
    ssize_t n;
    char recvline[MAXLINE];

    if (writen(sockfd, "\n", 1) != 1)
        throw runtime_error("failed to write to socket");
    if ((n = readline(sockfd, recvline, MAXLINE)) < 0)
        throw runtime_error("failed to read from socket");
    else if (n == 0)
        throw runtime_error("server terminated prematurely");
    if (sscanf(recvline, "%lf %lf", &x, &y) != 2)
        throw runtime_error("invalid data received from server");
    return true;
}
```

It starts by writing an empty line -- consisting of just a newline -- to the socket and reading from the socket. Normally, a read should return a line that contains a pair of doubles in ASCII form; if it reads 0 bytes, it is an indication of a problem with the server; and the read() call throws an exception. The protocol between the client and server is simple, ASCII and line-based. The are many reasons for this choice main ones being, portability, easy visibility on the sniffer in case of troubleshooting. Using a binary format or some form of serialization would add unnecessary complexity and create a layer of opacity between the user and the library; these are the same reasons for major Internet protocols like HTTP, FTP, SMTP and many more to count, being ASCII and line-based. Although there are calls to readline() and writen() functions, they will not be discussed again since these are the same functions utilized by the server and described above.

This concludes the discussion of the project. It is a lightweight and initial release, which was tested exclusively with both server and client running on the Linux platform.

References

1. Cygwin - a Linux-like environment for Windows - http://www.cygwin.com

2. MinGW - Minimalist GNU for Windows - http://www.mingw.org

3. BSD Sockets API - http://en.wikipedia.org/wiki/Berkeley_sockets

4. Windows Sockets API - http://en.wikipedia.org/wiki/Winsock

5. UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI, Prentice Hall, 1998, ISBN 0-13-490012-X.

6. POSIX Threads - http://en.wikipedia.org/wiki/POSIX_Threads