ECE 358, Spring 2014 — Assignment 4
Due Fri, July 4, 11:59:59 PM

(Use the drop box on Learn. Only one member of the group should submit, with both members' names on the submission in a README file.)

Implement an API for a reliable, connection-oriented, stream-oriented (RCS or rcs) protocol. You are given an API for a unreliable, connectionless, packet-oriented (UCP or ucp) protocol.

More specifically, you need to implement the calls we mention in Section 1. The only networking-related functionality you are allowed to use is the API that we discuss in Section 2. The implementation of that API is provided to you in Learn as `ucp.c`, which you should compile and use.

1 What you need to design and implement

You need to design and implement the following functions that constitute the API for a reliable, connection-oriented, stream-oriented protocol. Each should return -1 on error and set errno appropriately.

$int\ rcsSocket()$ — used to allocate an RCS socket. No arguments. Returns a socket descriptor (positive integer) on success.

$int\ rcsBind(int, const\ struct\ sockaddr\_in\ *)$ — binds an RCS socket (first argument) to the address structure (second argument). Returns 0 on success.

$int\ rcsGetSockName(int, struct\ sockaddr\_in\ *)$ — fills in the address information into the second argument with which an RCS socket (first argument) has been bound via a call to $rcsBind()$. Returns 0 on success.

$int\ rcsListen(int)$ — marks an RCS socket (the argument) as listening for connection requests. Returns 0 on success.

$int\ rcsAccept(int, struct\ sockaddr\_in\ *)$ — accepts a connection request on a socket (the first argument). This is a blocking call while awaiting connection requests. The call is unblocked when a connection request is received. The address of the peer (client) is filled into the second argument. The call returns a descriptor to a new RCS socket that can be used to $rcsSend()$ and $rcsRecv()$ with the peer (client).

$int\ rcsConnect(int, const\ struct\ sockaddr\_in\ *)$ — connects a client to a server. The socket (first argument) must have been bound beforehand using $rcsBind()$. The second argument identifies the server to which connection should be attempted. Returns 0 on success.

$int\ rcsRecv(int, void\ *, int)$ — blocks awaiting data on a socket (first argument). Presumably, the socket is one that has been returned by a prior call to $rcsAccept()$, or on which $rcsConnect()$ has been successfully called. The second argument is the buffer which is filled with received data. The maximum amount of data that may be written is identified by the third argument. Returns the actual amount of data received. "Amount" is the number of bytes. Data is sent and received reliably, so any byte that is returned by this call should be what was sent, and in the correct order.

$int\ rcsSend(int, const\ void\ *, int)$ — blocks sending data. The first argument is a socket descriptor that has been returned by a prior call to $rcsAccept()$, or on which $rcsConnect()$ has been successfully called. The second argument is the buffer that contains the data to be sent. The third argument is the number of bytes to be sent. Returns the actual number of bytes sent. If $rcsSend()$ returns with a non-negative return value, then we know that so many bytes were reliably received by the other end.

$int\ rcsClose(int)$ — closes an RCS socket descriptor. Returns 0 on success.

A server based on your API will invoke $rcsSocket()$, $rcsBind()$, $rcsListen()$, $rcsAccept()$, $rcsRecv()$, $rcsSend()$ and $rcsClose()$. A client will invoke $rcsSocket()$, $rcsBind()$, $rcsConnect()$, $rcsRecv()$, $rcsSend()$ and $rcsClose()$.

2 What you are provided

You are provided an API for an unreliable, connectionless, packet-oriented protocol, and its implementation. You are allowed to use only this API as the underlying networking functionality. That is, you are not allowed to instantiate or use standard UDP/TCP sockets for communication. (However, you are allowed to use other IPC mechanisms to implement the API – see Section 5 below titled, "Hints.")

The API comprises the following functions. A return of -1 from any of these calls indicates an error. The global variable errno is set appropriately.

$int\ ucpSocket()$ — returns a UCP socket descriptor. No arguments.

$int\ ucpBind(int, const\ struct\ sockaddr\_in\ *)$ — binds a UCP socket (first argument) to the address structure (second argument). Returns 0 on success.

$int\ ucpGetSockName(int, struct\ sockaddr\_in\ *)$ — fills in the second argument with the address information that is bound to a UCP socket via the $ucpBind()$ call. Returns 0 on success.

$int\ ucpSetSockRecvTimeout(int, int)$ — sets a timeout for each subsequent $ucpRecvFrom()$ call on the socket (first argument). The second argument is the timeout value, in milliseconds. If the second argument is 0, this is interpreted as ∞ (i.e., never timeout).

$int\ ucpSendTo(int, const\ void\ *, int, const\ struct\ sockaddr\_in\ *)$ — sends data from a buffer (second argument) whose number of bytes is indicated by the third argument. The last argument is the destination. The first argument is the socket on which to send. Returns the actual number of bytes sent. This mode of sending data is unreliable in that data that is sent may get lost or corrupted. In particular, if $ucpSendTo()$ returns a value v, fewer than v bytes may be received by the other end as a consequence of the invocation to $ucpSendTo()$.

$int\ ucpRecvFrom(int, void\ *, int, struct\ sockaddr\_in\ *)$ — receives at most as many bytes as indicated by the third argument into the buffer that is the second argument. The socket associated with the data reception is the first argument. The identity of the sender is filled into the last argument. This is a blocking call unless $ucpSetSockRecvTimeout()$ has been successfully called on the UCP socket beforehand with a timeout > 0.

$int\ ucpClose(int)$ — closes a UCP socket and returns 0 on success.

$int\ ucpSelect(\ldots)$ — a wrapper for *select*(), which can be used to check if a set of file descriptors is ready for reading and/or writing. See the man page for *select* for details. Specifically, whether a read/write would block. The call takes a time argument, which is an upper-bound on the time within which the call returns.

3 Makefile

You must provide a makefile as part of your submission. Your makefile must have the following two targets: clean, and librcs.a. We will exercise only those two targets. That is, we will issue "make clean," and then "make librcs.a." We will expect the first to remove all .o, executable and .a files from the current directory. We will expect the latter to create, from your source-code, librcs.a, which is an archive file that has all the RCS functions we discuss above. We will link our client and server implementations against this librcs.a that your makefile generates.

4 How we will mark

We will write a client and server that works using the standard socket API for TCP-based communication. An example is provided on Learn.

Our server may be multithreaded. That is, when $listen()$ returns, we will call $accept()$. We may then create a new POSIX thread using $pthread\_create()$ that then exchanges data with the client on the socket returned by $accept()$. Our client and server will send data back and forth using $send()$ and $recv()$. See our example tcp-client and tcp-server that work with one another.

We will then replace all calls to the standard socket API with the corresponding RCS calls (see Section 1). We will check whether the behaviour is the same, except for somewhat poorer performance as discussed below. We will set the value of pDoEvil in $ucpSendTo()$ to some value that we will not disclose beforehand. (But you can of course test your implementation with various values of pDoEvil.)

Performance We fully expect that the performance of your implementation, in terms of how long it takes to exchange data between a client and server, will be worse than with the standard socket API. This is because what underlies $send()$ is more reliable than $ucpSendTo()$, which underlies $rcsSend()$.

We will set a baseline for the performance as follows. All the data that we plan to send using your RCS API, we will first send as 150-byte packets using the UCP API. We will measure the time for all packets to arrive intact. That is, if some packets do not arrive, or arrive mangled or truncated, we will resend them. We will then multiply this time by 10. Your performance must be no worse than this value. For example, if we are able to send the data as discussed here within 1 second, with your implementation of the RCS API, we must be able to send the data within 10 seconds.

Coding Standard We expect you to follow the same guidelines as described in earlier programming assignment.

Grading Scheme:

a) 5% if you implemented something meaningful that compiles. You should carefully adhere to "What to submit" below, or you risk getting 0.

b) 90% if you implementation works when tested with our client and server (as tested using the standard socket API), but with a lower performance than expected. You will get points between 5% and 60% if you have some useful functionality, but it does not work fully.

c) 100% if, in addition, your program follows a coding standard and is well documented.

5 Hints

An RCS socket that is returned by $rcsSocket()$ does not have to be a conventional socket. It can be whatever you want it to be. (But of course, it has to be of type int.) For example, the integer you return can be an index into an array, where each entry of the array is of type a data structure that you define.

Of course, there is some mapping between an RCS socket and a UCP socket. But it does not have to be one-to-one. Let s be an RCS socket. You may, for example, create a UCP socket when s is created. That is, underlying $rcsSocket()$ is a call to $ucpSocket()$. However, you may choose to multiplex s and every socket that is returned by an $rcsAccept()$ on s onto the same UCP socket.

6 What to submit

A single zip file called `ece358a4.zip`. Unzipping this zip file should result in a single folder called `ece358a4`. All your source files must be within this folder. You're free to create subfolders within that folder. Your `ece358a4` (i.e., top-level) folder must contain a Makefile as discussed above in the section titled, "Makefile."

The ucp.c file on which you rely must be assumed to be in the top-level folder, `ece358a4` and not in any subfolder you may choose to create. We do not have to provide a ucp.c file as part of your submission. Whether you do or not, after unzipping your `ece358a4.zip`, we will move over our version of ucp.c into your `ece358a4` folder. If any exists, we will overwrite it.

We will check, during compilation, that our ucp.c is the one that's adopted. You should not turn in anything that contains a `main()` because this will break out build, and cause you to get an automatic 0.