CS 214: Systems Programming, Fall 2020
Assignment 3: Who's There?

0. Introduction

In this assignment you will make use of C sockets to implement a simple network service: a knock knock joke server.

Knock knock jokes are extremely formulaic and often rely on simple puns and malapropisms. While not the height of sophisticated humor, their request/response structure and limited range of manipulation strongly mirrors network service protocols and as such are a good basic introduction to the same.

Your task will be to write a knock knock joke server using the protocols established below. There is an application protocol and a message format. The application protocol specifies what messages of what type are allowed when. The message format specifies how the application messages should be constructed and read/written from/to your sockets.

Your server should be able to handle a single humor-devoid client at a time.

1. Background - Knock knock jokes

The basic format of every knock knock joke is:

Knock, knock.
Who's there?
<setup line>.
<setup line>, who?
<punch line><punctuation>
<expression of annoyance and/or disgust or surprise><punctuation>

The conceit of the format is apparently the taking advantage of the brief but plausible confusion of someone answering a door when an odd 'name' is given as an answer when a name is expected. By way of example:

Knock, knock.
Who's there?
Who.
Who, who?
I didn't know you were an owl!
Ugh.

Or:

Knock, knock.
Who's there?
Dijkstra.
Dijkstra, who?
That path was taking too long, so I let myself in.
Ahh!

2.a Implementation - KKJ Application Protocol

Given the strict structure of knock knock jokes, the KKJ application protocol consists of the below.

Notes and symbols:

all messages are character sequences, but are not sent as strings.
all messages are constructed using the KKJ Message Format (below).
values in double quotes (“ “) are character literals and must appear
  note this does not mean they are C strings, but a character list/array/sequence
  note this means double quotes are only tags, they are not data to be sent
values in angle brackets (< >) are variable, but required.
important events that are not messages are enclosed in parentheses (( )).
<expression of A/D/S> is an expression of annoyance, disgust or surprise

Server	Client
	(connect to server)
	(begin read)
(receive client connection)	
(send initial to client:)	
“Knock, knock.”	
	(receive initial message)
	(send response to server:)
	“Who's there?”
(recv response)	
(send setup to client:)	
<setup line>”.”	(receive setup line)
	(send response to server:)
	<setup line>”, who?”
(recv response)	
(send punch to client:)	
<punch line><punct.>	
	(receive punch line)
	(send response to server:)
	<expression of A/D/S><punct.>
(recv expression of A/D/S)	
(close/free)	
	(close/free/return)

Errors and Exceptions:

If ever a message is sent that does not match the requirements of the protocol, the side receiving the bad message should not send the next message but instead respond with an error message (see the KKJ messaging format below) indicating the error type and should shut down their context gracefully.

2.b Implementation - KKJ Message Format

There are certain constraints and issues when communicating over a network. While sockets are represented by file descriptors, read() and write() work a bit differently when interfacing with a network. Sockets are a little for most operating systems because they bridge the basic device types: character devices and block devices. Character devices deliver data byte by byte only and block devices deliver data in fixed-size chunks only. Networks deliver data in preconfigured chunks, whose sizes can change, one byte at a time. Since they don't fit in to a particular device type some care has to be taken when sending data via network.

The first three characters are the message type. REG denotes a regular message that should be read in, checked and if it holds to the requirements, responded to normally. ERR denotes an error message that should be read in and handled, however the protocol is now ended and the side receiving the error can deallocate and shut down. Once an error message has been sent, the sender should deallocate and shut down.

Regular messages:

A regular message is a message that should be read in, checked and if it holds to the requirements, responded to according to the KKJ application protocol.

Every message delivered that is not an error message should have the following format:

REG|<length segment>|<message segment>|

REG: the characters 'R', 'E', 'G' in sequence.
|: a single '|' character
<length>: the character representation of the length of the message segment
e.g. if the message is 23 characters long, then '2', '3' in sequence.
|: a single '|' character
<message>: the characters that make up the message, but not a C string.
|: a single '|' character

The message length includes only the length of the message segment and not the '|' separators.

e.g.:

To send the “Knock, knock.” message:
REG|13|Knock, knock.|

To send the “Who's there?” message:
REG|12|Who's there?|

Error messages:

An error message is a message that should be read in, checked and handled, however no response is sent. An error message halts the protocol and causes both the sender and receiver to deallocate and shut down after sending/receiving.

The default handling action is to output a textual description of the the error code on standard out.

Every message denoting an error should have the following format:

ERR|<error code>|

ERR: the characters 'E', 'R', 'R' in sequence.
|: a single '|' character
<error code>: one of the error codes below.
|: a single '|' character

Error codes:

M0CT - message 0 content was not correct (i.e. should be “Knock, knock.”)
M0LN - message 0 length value was incorrect (i.e. should be 13 characters long)
M0FT - message 0 format was broken (did not include a message type, missing or too many '|')

M1CT - message 1 content was not correct (i.e. should be “Who's there?”)
M1LN - message 1 length value was incorrect (i.e. should be 12 characters long)
M1FT - message 1 format was broken (did not include a message type, missing or too many '|')

M2CT - message 3 content was not correct (i.e. missing punctuation)
M2LN - message 2 length value was incorrect (i.e. should be the length of the message)
M2FT - message 2 format was broken (did not include a message type, missing or too many '|')

M3CT - message 3 content was not correct
(i.e. should contain message 2 with “, who?” tacked on)
M3LN - message 3 length value was incorrect (i.e. should be M2 length plus six)
M3FT - message 3 format was broken (did not include a message type, missing or too many '|')

M4CT - message 4 content was not correct (i.e. missing punctuation)
M4LN - message 4 length value was incorrect (i.e. should be the length of the message)
M4FT - message 4 format was broken (did not include a message type, missing or too many '|')

M5CT - message 5 content was not correct (i.e. missing punctuation)
M5LN - message 5 length value was incorrect (i.e. should be the length of the message)
M5FT - message 5 format was broken (did not include a message type, missing or too many '|')

Message Handling:

Since every message must start with either 'REG' or 'ERR', by default every message is at least three bytes long. Based on the message type, you will know the format to follow.

Since you know the locations of the separators, you can verify if the length value is correct.

Since you know the locations of the separators and can parse the length value given, you can have a decent expectation of how long the message ought to be.

Since '|' is not a character to expect in normal text, you can presume any '|' you see is a separator.

Be sure to check for punctuation characters, where required. Be sure to check for punctuation characters that are required to appear at the end of variable messages like the setup line, punch line, and the A/D/S expression. Missing punctuation results in a message content error, even though their contents are variable (i.e. the setup line could be anything, but must terminate in a period).

Be sure to check that the setup response includes the entire setup line as its initial content.

3. Operation

You are only required to build the server, however you will likely want some type of client to test it.

Your server should take a single argument (others are optional, see extra credit below) that is the port number it should start on. When testing your server, be sure to pick a port number above 5000 and below 64K (65536). Avoid port numbers that involve repetition or simple patterns like 11111 or 12345.

e.g.: ./KKJserver 31901

While testing you might encounter a bind/address in use error if you quit your server and start it back up again quickly. If this occurs, add one to your port number and use that. After approximately three minutes, your old port number will be released.

There is no command or interface to shut down your server. Shut it down with Ctrl+C.

After delivering a joke to a client, your server should go back to accept() the next.

Your serve will be delivering the setup and punch lines. By default you only need to support a single joke, so this can be hard-coded. (see extra credit for other options)

Be sure to check everything; number of arguments, port number for valid range, number of bytes read in a message.

Do not presume that one read will get the entire message at once. If you haven't seen a separator or you have not read all the bytes you were promised (in the length field) there are supposed to be more on the way.

4. What to Turn In

A tarred gzipped file named Asst3.tgz that contains a directory called Asst3 with at least the following files in it:

Asst3.c:	your main
Makefile:	must have both an “all” and “clean” directive all should build an executable named “KKJserver” and be first clean should remove any files automatically constructed by the Makefile
testcases.txt	a brief discussion of your testcases. Be sure to include at least five.

Your code will be invoked as: ./KKJserver <port number>

e.g.: ./KKJserver 41910

5. Grading

Correctness - how well your code operates
Testing thoroughness - quality and rationale behind your test cases
Design - how well-written and robust your code is, including modularity and comments

6. Extra Credit

10pts: joke file

Take as a second argument to your program a file that holds a joke list. The joke list should be newline-separated setup and punch lines.

e.g.:
Who.
I didn't know you were an owl!

Dijkstra.
That path was taking too long, so I let myself in.

Each client that connects should get a random joke from your joke file. It would be a good idea to read them all in when you start up in order to avoid disk IO while talking to the network.