UNIX socket spellcheck service

Maciej Domagalski

The premise is simple

- The server awaits for client requests
- The client sends a word to spellcheck
- The server checks if that word is in the dictionary
- If it's a word, the server responds affirmatively
- If it's not, server looks for whatever's the closest, and suggests that word
- If the server can't find anything similar to the requested word, it responds negatively

The server program

- Before opening any socket, the dictionary is loaded from a large text file to a regular array (which must be created on the heap!)
- Then the main part starts, with opening an UNIX socket (using SOCK_STREAM, because reliability is more important than speed)
- The socket is bound to a given address, which is a UNIX path
- We're now ready to listen for clients

Opening socket and preparing the address

```
int openSocket(){
        // open the socket
        printf("Opening socket\n");
        int sock = socket(AF_UNIX, SOCK_STREAM, 0);

        if(sock < 0){
                perror("Couldn't open socket");
        exit(1);
        }
        printf("Socket open\n");
        return sock;
}

struct sockaddr_un prepareSocket(){
        struct sockaddr_un servinfo;
        servinfo.sun_family = AF_UNIX;

        strcpy(servinfo.sun_path, ADDRESS);

        return servinfo;
}

// main function
int sock = openSocket();
struct sockaddr_un servinfo = prepareSocket();
```

```
#define ADDRESS "/tmp/spellSock"
```

Binding the socket

```
// main function
unlink(ADDRESS); // important!
if (bind(sock, (struct sockaddr *)&servinfo, SUN_LEN(&servinfo)) == -1){
	perror("Couldn't bind socket\n");
	exit(2);
}
printf("Socket bound\n");
```

The main loop

- The server waits for clients to accept
- Upon accepting, the server receives a word to spellcheck
- The word is being searched for in the dictionary array
- If it's found, the server gets straight to the response – simply sends "Correct" to the client and closes the connection
- If the word isn't found, a separate array is created, filled with Levenshtein distances to every word (the smaller it is, the closer we are to the provided word)
- That new array is searched through in order to find the smallest distance, which corresponds to a potential correction
- If that distance is greater than 3 (chosen arbitrarily), the server sends a negative response – whatever was sent, was not even close to a real word.
- If that's not the case, the client receives the corrected word
- The connection is closed, and the cycle repeats

Main loop (stripped to keep concise)

```
while(true){
        Int newsock = accept(sock, &client_info, &client_addrlen);

        // receive the word
        int rbytes = 0;
        char word[31];
        if ((rbytes = read(newsock, word, 30)) > 0){
                word[rbytes] = 0;
        }

        // create response buffer
        char responseBuffer [100];

        // process the word
        bool correct = wordSearch(word, wordsTable);
        if(correct)
                sprintf(responseBuffer, "Correct\n");
        else{
                findCandidate(word, responseBuffer, wordsTable);
        }

        // send response to client and close socket
        write(newsock, responseBuffer, strlen(responseBuffer) + 1);
        close(newsock);
}
```

The client program

- A lot of clients can be written for this service, but I'll present a very simple one, that can be easy to follow
- The client reads a word to spellcheck from a command line argument
- The socket is opened in the same way as in the server program
- We connect to a server using the same address
- Upon connecting, the client sends the word to the server and waits for a response
- After closing the connection from a server side, the client closes the socket and the program terminates.

Connecting to server

Opening the socket is the same as in the server program.

```
if(connect(sock, (struct sockaddr*)&servinfo, SUN_LEN(&servinfo)) == -1){
    close(sock);
    perror("Failed to connect");
    exit(3);
}
```

Client program – the main part

```
// send the word
write(sock, word, strlen(word) + 1);
printf("Sent word: %s, waiting for response\n", word);

// receive the message
int rbytes = 0;
char responseBuffer[100];
if ((rbytes = read(sock, responseBuffer, 100)) > 0){
        responseBuffer[rbytes] = 0;
        printf("%s", responseBuffer);
}
printf("\n");

// close the socket and exit
close(sock);
return 0;
```

A lot of stuff can go wrong on the server...

- The socket can error out when trying to open for a number of reasons (like invalid flags, permissions, unsupported types when UNIX is mixed with INET sockets, etc.)
- Binding will fail, if the address file exists (like when a TCP port is used, you can't open it twice)
- Finally, accepting the client in a wrong way can crash the entire server (you don't want that, trust me)

…and on client's side

- Same problems with opening sockets apply
- Connecting can fail (like when the server isn't running, well duh...)
- What if the word isn't provided? We don't want to send a string that doesn't exist, so we'd better error out as soon as we figure out that argument count isn't correct
- Finally, the byte string can be read backwards – and then what was supposed to say "apple" might end up as "6e!��"

Useful links

- You can read through the code and see my struggles at https://github.com/Macdom/spellcheck (commit history shows how I was changing the concept – tries, INET sockets and more ideas have been dropped)
- Dictionary file: https://github.com/dwyl/english-words
- Levenshtein's algorithm was taken from WikiBooks: https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#C