Operating Systems

Exercise H: GUIs and Eventing

In the previous exercises we built a network chat and a GUI for it. Now it's time to combine them into one: A GUI based network chat.

1. Gluing time

In this step we'll collect all we need. Putting all the source code into one C file. The program won't work after this step and we'll do the rewiring of the pieces later on. This step is about collecting all the pieces.

- Download gui.a and gui.h again from the personal schedule.
- Create a new file gui_net_chat.c.
- Take the chat GUI source code from last exercise (gui_chat.c) and copy it into gui_net_chat.c.
- We only need parts from the network chat (chat.c). Take the source code of the following 3 functions and copy them into gui_net_chat.c. You have to insert them before the GUI functions.
  - sockaddr_from_str()
  - wait_for_first_connection()
  - establish_connection_to()

  Also take all the #include lines from the console chat program and put them at the start of gui_net_chat.c. That way the compiles knowns about all the operating system functions we used in the chat code. You don't need the same #include line twice.
- Compile the program like the GUI programs from the previous exercise. This is just to see if the compiler is happy with how the code was copy & pasted together (no syntax errors, no #include lines missing, etc.).

  ```
  gcc -Wall -std=c99 gui_net_chat.c -o gui_net_chat gui.a $(ar -p
  gui.a LDLIBS)
  ```

2. Hooking up the client

We've got all the pieces in one program now. The GUI code works just fine but the network code just sits there, doing nothing, never being called. In this step we wire up some of the network code so our GUI program can connect to a server.

- T When the user presses the “start client” button our client_start_cb() callback is called. So in there we'll put our code to connect to a server.

  Get the current values of the address and port text fields and store them in local string variables. In order to access those text fields you'll have to make the Ihandle* variables that store the text controls themselves global variables. Just like with the message text field in the previous exercise last week.

- T Use the sockaddr_from_str() function to convert the address and port strings into a binary network address. The main() function of our old chat program did the same (hint, hint, you don't have to write it again).

- T Use establish_connection_to() to connect to that address. Store the returned file descriptor in a new global variable called “connection_fd_to_send_messages”. If everything works we're now connected to the server. We did all we have to do when the user presses the “start client” button.

- T When the users presses the “send” button right now we “only” add the message to the local chat history. All this happens in the send_message_cb() callback. Now we also want to actually send the message over the network when the user presses the “send” button. To do that use write() to write the message string into the connection_fd_to_send_messages file descriptor. With that call our message is on the way.

- T Time for testing: Run the command line chat server in another terminal or find a partner that'll run the server for you. Then test your GUI network client by connecting to that server and sending a message.

  Note: Right now you can only send messages. We haven't hooked up the code to receive messages. So don't worry that the server can't send you messages, yet. ;)

3. Receiving messages

This is where the good stuff happens. In the previous step we connected to a server and got a file descriptor that represents this connection. When data arrives on that file descriptor we got a message and should display it in the chat history.

In our old console chat program we used poll() to tell the operating system that we want to wait for data from the connection file descriptor. But this time our network code runs in a GUI program. And remember: In a GUI program the GUI library is in control of the main loop.

So instead of telling the operating system directly via the poll() function we have to tell the GUI library that we want to do something when data arrives on the connection file descriptor. Thats what the IupLoopAddWatch(int fd, short events, IWatchCallback fd_ready_cb) is for. So here we go.

- As with all GUI code we first need a callback. This function should be called when data on our connection file descriptor arrives. Since this is the first time you'll get one piece of code for free. Add it to your GUI functions:

  ```
  int receive_message_cb(int connection_fd, short revents) {
      printf("TODO: React to new data on network connection\n");
      return IUP_CONTINUE_WATCH;
  }
  ```

- We've got the callback. We open the network connection in client_start_cb() so we also should tell the GUI library right there what to do with the connection. Add the following code after your call to establish_connection_to().

  ```
  IupLoopAddWatch(connection_fd, POLLIN, receive_message_cb);
  ```

  Replace “connection_fd” with the variable you used to store the file descriptor of the network connection. This function call tells the GUI library IUP that whenever data is ready (POLLIN, remember?) on the network connection that it should call our receive_message_cb() callback function.

- Now the events are wired up, time to react to them. receive_message_cb() is called as long as data is ready on the network connection. So in that callback we have to do what we previously did in the main() function of our console chat program.

  To start off read data from the network connection into a local 1024 byte buffer.

- Output the read data by using the "APPEND" attribute of the chat history text field. This is the same as in the old poll() based network chat where we reacted to data from the network connection. But instead of writing to STDOUT_FILE_NO we put the text into the chat history.

  Tip: Use the code

  ```
  IupSetStrf(history_text_field, "APPEND", "message: %.*s",
  (int)bytes_read, buffer);
  ```

  to append a block of data that is bytes_read long to the chat history. The data in buffer doesn't have to be null terminated for that to work (btw. this trick also works with printf()).
- Add IupSetAttribute(history_text_field, "SCROLLTO", "1000000000,1000000"); right after appending the text to make sure we always see the latest message in the chat history.
- Testing time again: Run a command line server in another terminal and connect to it with your GUI chat program. Both sending and receiving messages should work now.

4. Hooking up the server

Almost all of the buttons in our GUI program now do what they're supposed to do. To finish the job we only have to make the “start server” button do it's job. It should actually start the server. That's why we also added the wait_for_first_connection() to your code. We just need to call it at the right time.

- When the “start server” button is pressed the server_start_cb() callback is called. A part of what we need is the same as we did to start the client: As before fetch the current values of the address and port text fields and store them in local string variables. Also use the sockaddr_from_str() function again to convert the address and port strings into a binary network address.
- Then call the wait_for_first_connection() function to wait until someone connects to us.
- Store the returned file descriptor in the connection_fd_to_send_messages global variable. This way send_message_cb() will send messages we type via that file descriptor.
- Also use IupLoopAddWatch(connection_fd, POLLIN, receive_message_cb) again to tell the GUI to call receive_message_cb() when data is ready on the network connection.
- Test it by running the GUI program to start a server. While that program is waiting for a connection start the GUI program again (in another terminal, by simply double clicking it in the file browser or by finding a partner). Use that program to connect to the server. Now you should be able to send messages between those two programs.

  You might notice that the server program is unresponsive until someone connects. Yep, that's broken. If you still got time you can fix with the next step below.