TPRG 2131 Project 2: Pi-2-Pi Secure Communications

Release 1.0: November 19, 2019

In this project, you and a partner are developing a super secure communication[1] system using Raspberry Pi microcontroller kits to build custom terminals to communicate with each other. The communication is peer-to-peer between the Raspberry Pi peers, but they communicate with the server to log into the system and to get information. Once the communication between Pi terminals is established, the server is idle.

This project can be developed in stages. Encryption is desirable but adds a layer of complexity therefore it can be left out initially. The most important is the communication between the client (Raspberry Pi terminal) and the server. As the client-server interaction progresses, the client moves through several states. The demonstration will be considered successful if the terminals manage to connect for one round trip exchange: “Hello, are you there?”, “Yes, here I am!”.

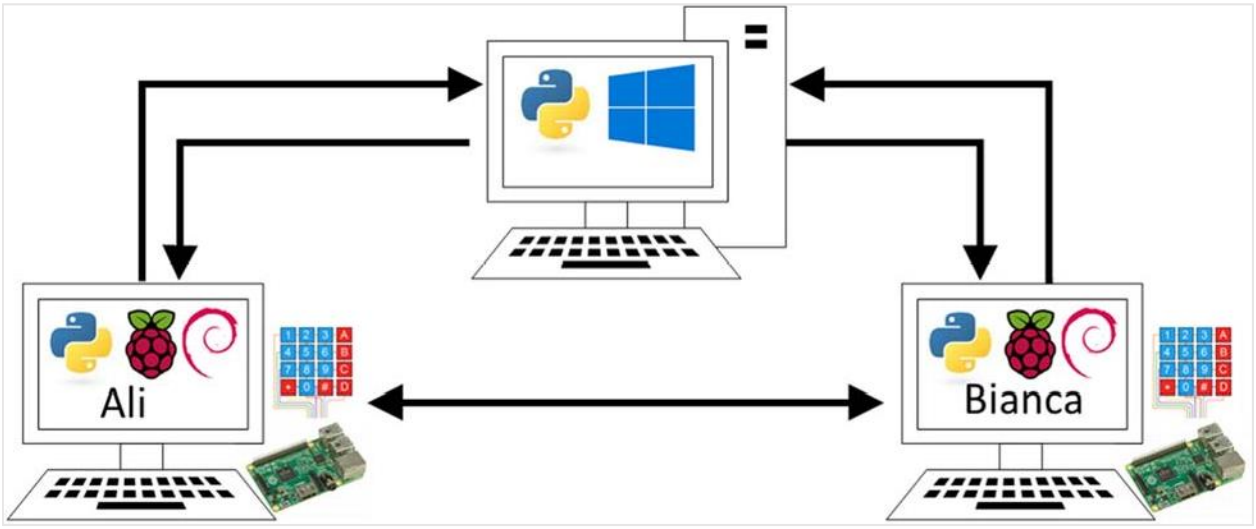

Figure 1: Communication system overview

Hardware

The secure terminal consists of the Pi, monitor and mouse, with a 4×4 keypad that will be used to enter the user’s passcode. The server is simply the Windows PC (or a student owned laptop). The terminals and server are connected to each other over the IP network (wired or wireless).

[1] No, not really. Security is hard, super security is harder. See Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno (2010). “Cryptography Engineering Design Principles and Practical Applications”. John Wiley & Sons., or read Bruce Schneier’s blog: https://www.schneier.com/

Typical session

Both users must first log into the system. Each sends an Authentication Request to the server, including the user ID and passcode. The passcode was entered on the 4×4 keypad attached to each Pi. The server replies with an Authentication Reply granting them access, in the case of a passcode match, or denying access otherwise. When a user is logged in, the server remembers that user's IP address. After logging in successfully, each terminal opens a socket to listen for incoming connections on port 8085.

A typical session begins by user A (Ali) at one terminal wishing to communicate with user B (Bianca) at her terminal. Ali sends a Lookup Request to the server asking for Bianca's address. If encryption is implemented, the server also gives Ali the encryption key for Bianca.

Next, Ali's terminal attempts to `socket.connect(`*`ADDRESS`*`)` a TCP connection to Bianca's address on port 8085. If Bianca is logged in and her terminal is `socket.accept()`-ing connections, Ali's terminal sends a Connect Request. Bianca must make sure that the connection request is legitimate. Bianca sends a Lookup Request to the server to verify Ali's IP address and (optionally, if implemented) Ali's encryption key.

Once Bianca has verified Ali's connection request, Bianca sends a Connection Reply to Ali. Finally, the communication proceeds. Each terminal goes through an infinite loop of `input()`→`send()` and `recv()`→`print()`.

The communication ends when either user types CTRL-C to end the chat session. After the end, each terminal continues to listen for connection requests to begin another chat session.

A typical session session might look like this. Here ali talks to bianca then types CTRL-C to end the connection (what ali types is in *italics*).

```
Login: ali
Enter passcode on keypad.
Logged in.
Enter destination user ID: bianca
Connection established, type your messages
Hello, how are you today?
I am fine, how about you?
I am well. Goodbye.
CTRL-C connection ended
Enter destination user ID:
```

In another session, ali is logged in and receives a connection request from bianca who then types CTRL-C to end the connection (what ali types is in *italics*).

```
Login: ali
Enter passcode on keypad.
Logged in.
Enter destination user ID:
Connection received from user bianca, type your messages
Hello, how are you today?
I am fine, how about you?
I am well. Goodbye.
Connection ended
Enter destination user ID:
```

Deliverables

This assignment is to be submitted in three parts:

1. The client statechart in the simplified UML statechart scheme introduced in week 4 as a PowerPoint or Visio document, or you may also use an online drawing tool like lucidchart.com (it makes you sign up for a free trial account[2]).
   No hand drawn or photographed statecharts will be accepted.
2. The Python source code files for client and server, each in a sub-folder, committed to your group's Fossil repository on server `tprg.set.org`. Grading will be based on correct operation, generally well organized layout and intelligent use of the SCM.
3. A demonstration of the system in actual operation in class during week 13 or outside of regular class times by appointment.

Grading scheme (25 points)

15	Functionality	Does it fulfill its functional requirements?
2	Style, heading & comments	Conformance to the course style guide.
3	Technique to be used	Does the program use the specified techniques?
1	Use of Fossil	Committed to your shared project Fossil repository, with reasonable commit messages. The final commit message should indicate that the files are ready to mark.
4	Statechart	Statechart on Visio, PowerPoint or other drawing tool (not hand drawn, no photos).
(5)	*Optional cryptography*	*Up to 2 bonus points for reasonably convincing passcode hashing; up to 3 bonus points for reasonably convincing encryption of messages and chat exchanges. This is a discretionary mark.*

If either client or server program crashes on start-up, the maximum functionality mark is 40% of 15 = 6 points. The code and statechart evaluation is separate.

Requirements

Passcode

The user must use the keypad to enter a numeric passcode, consisting of one or more digits 0—9. The 'F' key serves as the "Enter & Validate" key. The passcode is never displayed, not even "*" characters that would "leak" the number of digits.

Optional: The 'A' key is Backspace and the 'B' key is Cancel.

[2] lucidchart.com maintains its "cloud" server in the United States and any information including your login ID and password are subject to undisclosed search under the USA PATRIOT Act. You are not required to sign into this service to complete this project.

Communications protocol

The chat session must start with several phases to log in to the system, request user information and request a chat connection before any messages can be exchanged. The chat client goes through several states during the phases of the connection.

Information exchange between server and client is done with JSON formatted messages. Each message is a sequence of *name:value* pairs. Message types are identified with the msgtype key present in all messages.

Authentication phase

Each client sends an Authentication Request to the server. The authentication request includes the user ID and the hashed passcode. The server looks up the user ID and compares the hashed passcode to the hashed passcode it has stored. If the hashed passcodes match, the server returns an Authentication Reply message and also stores the current IP address of the user who has just authenticated.

The client sends an Authentication Request message to the server (Listing 1).

- msgtype: “AUTHREQ”
- userid: string as entered by the user making the request;
- passcode: the user’s passcode entered by the user on the keypad, then salted and hashed (see passcode hashing section).

Listing 1: Authentication Request message example

```
{
  "msgtype": "AUTHREQ",
  "userid": "ali",
  "passcode": "0023dc6bc244d474d2237ab53d256430b14eabe6db"
}
```

The server sends the Authentication Reply message to the client telling the client if the user is authenticated or not. The value of the *status* key is either “GRANTED” if successful (Listing 2), or “REFUSED” otherwise (Listing 3).

- msgtype: “AUTHREPLY”
- userid: string as entered by the user making the request;
- status: “GRANTED” if the credentials were accepted, “REFUSED” otherwise.

Listing 2: Authentication Reply message example – successful authentication

```
{
  "msgtype": "AUTHREPLY",
  "status": "GRANTED"
}
```

Listing 3: Authentication Grant message example – authentication refused

```
{
  "msgtype": "AUTHREPLY",
  "status": "REFUSED"
}
```

Lookup phase

The initiating client sends a Lookup Request to the server. The request includes the user ID of the initiator and the user ID of the person the initiator wishes to contact. The server replies with a Lookup Reply message giving information about the person, or no information if that person is not logged in. As a precaution against requests by users not logged in, the server also replies with no information if the user making the request is not logged in. The server replies with a Lookup Reply message. The reply includes the user ID of the person the initiator wishes to contact, the IP address of her/his client and (optional) the encryption key to be used when communicating with that person.

The client sends the Lookup Request message to the server (Listing 4).

- msgtype: “LOOKUPREQ”;
- userid: string as entered by the user making the request;
- lookup: the user ID of the person that the user is calling.

Listing 4: Lookup Request message example

```
{
  "msgtype": "LOOKUPREQ",
  "userid": "ali",
  "lookup": "bianca"
}
```

The server sends the Lookup Reply message to the client, with “status”:“SUCCESS” and information if found (Listing 5), or “status”:“NOTFOUND” and empty information fields otherwise (Listing 6).

- msgtype: “LOOKUPREPLY”;
- status: “SUCCESS” if the user ID is logged in, “NOTFOUND” otherwise
- answer: the user ID of the person that the user is calling;
- address: the IP address of the remote user;
- encryptionkey: the encryption key as a string.

Listing 5: Lookup Reply message example – User found, information returned

```
{
  "msgtype": "LOOKUPREPLY",
  "status": "SUCCESS",
  "answer": "bianca",
  "address": "192.168.1.23",
  "encryptionkey": "1234"
}
```

Listing 6: Lookup Reply message example – User not found

```
{
  "msgtype": "LOOKUPREPLY",
  "status": "NOTFOUND",
  "answer": "",
  "location": "",
  "encryptionkey": ""
}
```

Connection phase

The initiating terminal sends a Connection Request to the destination terminal. The request includes the user ID of the initiator. The destination terminal performs a Lookup Request of its own to the server and, if the address of the initiating user's terminal matches the one registered with the server, the destination terminal replies with a Connect Reply "accepted" message to go ahead with the chat session, otherwise it sends a Connect Reply "refused" message.

The initiator sends the Connect Request message to the destination terminal (Listing 7).

- msgtype: "CONNECTREQ";
- initiator: the user ID of the person making the request.

Listing 7: Connect Request message example – Ali requests connection to Bianca

```
{
  "msgtype": "CONNECTREQ",
  "initiator": "ali"
}
```

The destination sends the Connect Reply message "status":"ACCEPTED" to the initiator terminal (Listing 8), or "status":"REFUSED" otherwise (Listing 9).

- msgtype: "CONNECTREPLY";

- answer: “ACCEPTED” or “REFUSED”.

Listing 8: Connect Reply message example – Connection request accepted by destination

```
{
  "msgtype": "CONNECTREPLY",
  "status": "ACCEPTED"
}
```

Listing 9: Connect Reply message example – Connection request refused

```
{
  "msgtype": "CONNECTREPLY",
  "status": "REFUSED"
}
```

Chat phase

No special messaging format is required in the chat phase. The two-way connection is established between terminals and the main loop can simply read from the open connection as if it was a stream of bytes. Each client can “catch” the CTRL-C `KeyboardInterrupt` to end the chat at any time. No special protocol is needed to end a session.

Fossil

All source files must be submitted through the shared Fossil repository for your group. The files must be organized into two subfolders: terminal and server. Each commit must be done using your own user ID and password.

Cryptography

The extra functionality of hashing the passcodes and encrypting/decrypting the messages and chat exchanges is left as an option. A bonus of up to 2 points will be awarded for reasonably convincing hashing of the passcodes. A bonus of up to 3 points for reasonably convincing symmetric encryption of the messages and chat exchanges using the `cryptography` module in Python (see the last page of this document).

Implementation suggestions

This section gives some suggestions to implement the system however there are many possible approaches. See the “Incremental development” section below for partial implementation of the system.

Terminal state machine

The terminal program should be built as a state machine that changes state to manage each phase of the communication session (Figure 2). Each state transition is based on user commands to connect or connection requests from another terminal.

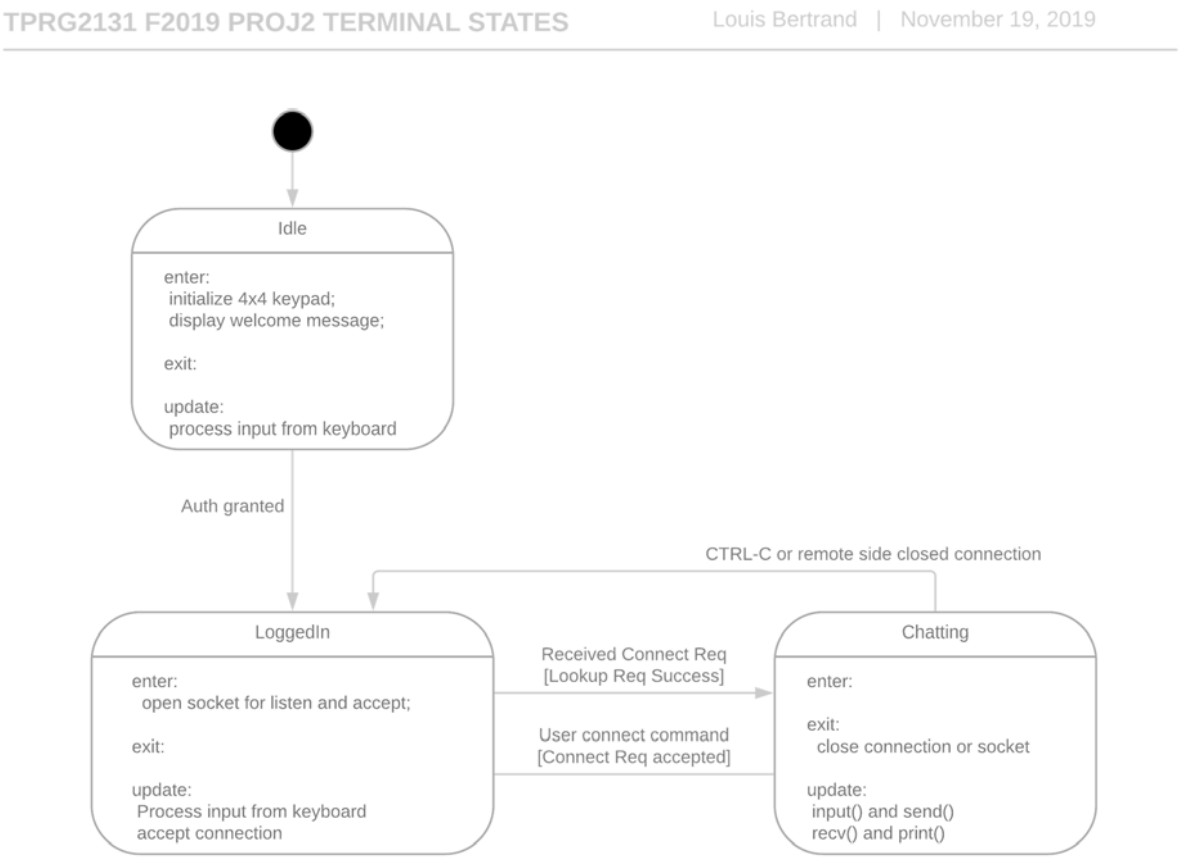

Figure 2: Statechart for one possible client state machine

Stateless server

The server can be implemented as a single main loop to `accept()` connections, process login and lookup request messages and update the logged-in table. See the `dummyserver.py` example.

Server user tables

The server must maintain two look-up tables of information about users. The first, `credentials`, is for user IDs and passwords (Listing 10), and the second, `logged_in`, is for user IDs who are logged in. Each table is implemented as a Python dictionary.

Credentials: the key is the user ID as a string (e.g. ali, bianca, fred,…) and the content is the passcode. Optionally, the passcode is hashed and the encryption key for that user is stored. The dictionary is initialized when the server starts. Optionally define the dictionary in a separate file and import it at start-up.

Listing 10: Server credentials dictionary; each user has passcode and encryption key.

```
credentials = {
    "ali": ["00123", "crypto"],
    "bianca": ["22456", "secret"],
    "fred": ["98765", "noclue"]
    }
```

Logged in: After accepting a user's credentials, the server creates a new *key:value* pair in the `logged_in` table for that user.

```
logged_in[new_user] = incoming_address
```

Server replies to requests

Authentication Requests: The server looks up the user ID in the credentials dictionary.

If the key is found in credentials,
    If the passcodes match,
        Format a Authentication Grant message and send it
    Else
        Format a Authentication Refused message and send it
Else
    Format a Authentication Refused message and send it

Lookup Requests: The server looks up both the requestor and destination user IDs in the logged_in dictionary.

If the requestor is found in logged_in AND the destination is found in logged_in,
    Format a Lookup Success message and send it
Else
    Format a Lookup NotFound message and send it

TCP ports

The server should listen on port 8082 (same as the in-class examples).

The Raspberry Pi terminal should listen on port 8085.

JSON messages

The JavaScript Object Notation standard maps almost directly to a Python dictionary. Dictionaries consist of an unordered list of *name:value* pairs and, when dictionaries are printed, they look almost the same as the equivalent JSON formatted string. The only real complication is encoding/decoding the JSON bytes in UTF-8 format to/from Python 3 Unicode strings (Listing 11).

Listing 11: Receiving a JSON string, decoding from UTF-8 and loading the JSON object into a Python dictionary

```
data = conn.recv(1024)  # Receive bytes from other side
if not data:  # Quit if no data – connection was ended
    break
recv_string = data.decode("UTF-8")  # decode from UTF-8 to String
print("Data:", recv_string)  # Have a look (debug only)
recv_dict = json.loads(recv_string)  # Load into a dictionary
```

Unknown or unimplemented keys

As a general rule, any program processing a JSON message ignores *key:value* pairs that it doesn't know about. In other words, if the program finds an unexpected key, it is not an error. Simply ignore it, as it may be processed elsewhere, or be for an optional feature not yet implemented.

No logging out

To limit the complexity of the project, there is no logout in this system. If you wish to implement the logout, add a logout request message to your system to remove users from the logged_in table. You may also remove users from the table if there was no activity for a certain timeout value.

Test server for terminal client requests

If you have not yet developed a server ready to respond to login and lookup requests, you can use the test server file `dummyserver.py` running on your own Windows or Linux system to test as you develop your Raspberry Pi terminal. Run the test server from the command line, either Windows PowerShell or CMD, or Linux shell (bash).

A test server will be running (at least during class time) on `tau.durhamcollege.net` [205.211.181.143] listening on port 8082 to respond to terminal requests from the Raspberry Pi client.

Authentication Requests: The test server will always reply with the “GRANTED” message to an authentication request by userid “ali”, and refuse the request from any other userid. The server ignores the password – it could be any string or even the empty string "".

Dummy authentication request:

```
{
  "msgtype": "AUTHREQ",
  "userid": "ali",
  "passcode": "00123"
}
```

Dummy authentication granted, but only to “ali”:

```
{
  "msgtype": "AUTHREPLY",
  "status": "GRANTED"
}
```

Any other userid will result in the “REFUSED” message.

```
{
  "msgtype": "AUTHREPLY",
  "status": "REFUSED"
}
```

Lookup Requests: The test server will always reply with the “SUCCESS” message to a lookup request by userid “ali” to lookup “bianca”, and return “NOTFOUND” for any other combination of userid and lookup.

Dummy lookup request:

```
{
  "msgtype": "LOOKUPREQ",
  "userid": "ali",
  "lookup": "bianca"
}
```

Dummy lookup successful, but only to “ali” for “bianca”:

```
{
  "msgtype": "LOOKUPREPLY",
  "status": "SUCCESS",
  "answer": "bianca",
  "address": "192.168.1.23",
  "encryptionkey": "1234"
}
```

Any other combination of userid and lookup will result in “NOTFOUND”.

```
{
  "msgtype": "LOOKUPREPLY",
  "status": "NOTFOUND",
  "answer": "",
  "location": "",
  "encryptionkey": ""
}
```

Test client for server development

If you have not yet developed a terminal application that is able to make login and lookup requests, you can use the test client file `dummyclient.py` running on your own Windows or Linux system to test as you develop your Windows server.

The client file makes five requests to exercise the options given above in the test server test cases then exits.

Passcode entry from the 4×4 keypad

Use the example files from week 9 on DC Connect. Split off the keypad entry function as a worker function in a thread and read the characters one by one from a queue until the 'F' character is received. Assemble the passcode as a string of characters representing the digits (it's a string, not an integer).

Passcode hashing

The traditional Unix password hashing algorithm encrypts the password with a hash function[3]. However simply hashing the password provides some predictability that could be used to guess the password, especially simple ones like "123". To circumvent this weakness, the authors of the original scheme made the hashed value less predictable by prepending a two character "salt" value to the password before hashing it, then prepended the salt to the "salted" hash before storing it (Alan Schwartz, Gene Spafford, and Simson Garfinkel 2003). One reasonable source of salts would be the milliseconds and tens of milliseconds digits from a call to `time.monotonic()`.

Use the Python `hashlib` library of strong cryptographic hashes (based on the widely used OpenSSL library). Most of the SHA-2 or SHA-3 family of algorithms may be used (US National Institute of Standards and Technology 2015). The example file `passcode.py` uses the SHA-256 hash function. The `hexdigest()` method returns a string of hexadecimal digits that represents the hashed value.

Encrypting messages and the chat exchange

Use the `cryptography` module in Python. A complete discussion of cryptography is beyond the scope of this assignment.
See the module documentation:
https://cryptography.io/en/latest/

[3]Python has a built-in `hash()` function but it returns a signed integer. If the returned integer is negative it is printed with the – sign which is hard to deal with. Python doesn't have an unsigned integer type (unlike C or C++). Using the `hashlib`module is simpler and safer since it uses algorithms vetted by experts.