University of Mississippi, Department of Electrical Engineering

El E 425 — TerminalTalk

Bryan Harper

November 26, 2016

Contents

Part I

Exegesis

1 Introduction

1.1 Overview

The purpose of this project is to design and build a distributed network application utilizing the relevant principles of computer networking. The project begins in section 1.2 with the conceptual design of a terminal based chat application called *TerminalTalk* . After a clear picture of the application is obtained, the focus shifts in section 2 to an exegetical discussion of the application layer in computer networking. In sections 2.1.3 and 2.2.3 the conceptual design and theoretical discussion will be compared to decided how to implement the application. Finally, in section 3 the application code itself will be developed. There is also an API for the code included in Part II of the paper.

1.2 Conceptual Design

In order to determine the application architecture and protocol with which to implement the software it is first necessary to develop a clear picture of the expected end product. The following is a description of what an end user will experience when running the *TerminalTalk* application.

The application will be completely run from the command line (terminal on Linux). There will be no additional graphical user interface beyond what the terminal provides (ANSI characters, cursor animation, colors, etc.). Thus, any interaction the user has with the application will be through her keyboard, in the form of standard input or specialized commands.

Upon starting the application—with the option of specifying a moniker (username)—the user will be connected to a *chat room*. The user's arrival will be announced to all users currently connected:

```
[moniker] has entered.
```

There will also be available to the user a non-intrusive prompt for the user to deliver a message to the room:

```
Talk:
```

If the user types a message and then presses `'Enter'` on their keyboard, the user will simply see their message displayed after the prompt, and then another prompt on a new line.

```
Talk:  [message inputed by user]
Talk:
```

However, any other user in the room will see something like this:

```
[moniker1]:  [message inputed by user1]
Talk:
```

Note that the first user's moniker is colored. When a user first begins the application the program randomly assigns a color to the user. However, if a user wishes to change her assigned color, she can use the command `\setcolor`:

```
Talk:  \setcolor green
Your color has been set to green.
```

There is also a command to change the user's moniker:

```
Talk:  \setmoniker Alan
Hello Alan.
```

And a command to disconnect:

```
Talk:  \disconnect
You have disconnected.
```

Seen from the perspective of other connected users, the output resulting from a user running the above commands might look like this:

```
Anonymous has entered.
Anonymous:  Hello everyone!
Anonymous:  This is Alan.  Let me change my moniker.
Alan:  There we go.
⋮
Alan:  I've got to run!  Later.
Alan has exited.
```

In summary, there are a few general requirements evident from the above:

1. The chat room is expected to always be available for connection to the user.
2. Once a connection is established with the chat room, the connection must be maintained until the user disconnects.
3. It is expected that a user's messages will be conveyed accurately
4. It is important that the order in which messages are sent is preserved.

In case those requirements are not yet obvious, there will be further discussion of them in sections 2.1.3 and 2.2.3.

2 Theoretical Discussion

2.1 Application Architecture

Network applications are processes distributed over multiple end-systems. In designing a network application, then, it must be decided how these various processes stand in relation to one another. Do all the processes, for example, perform essentially the same tasks in response to one another? Or, is there a small subset of processes with specialized functions? The answers to these questions constitute the notion of an *application architecture*.[2] There are any number of possible alternative architectures but two are dominant:

> In choosing the application architecture, an application developer will likely draw on one of the two predominant architectural paradigms used in modern network applications: the client-server architecture or the peer-to-peer (P2P) architecture. [1]

This project narrows itself to these two options—i.e., peer-to-peer and client-server. Each architectural paradigm will be discussed and then compared to the requirements of *TerminalTalk* .

2.1.1 Client-Server Architecture

In a client-server architecture the processes running on different end-systems are divided into two categories, servers and clients. A server is at the center of the distributed application and generally there is only one (or few) server(s) but numerous clients. Thus, although applications utilizing the client-server model are distributed, things are strongly centralized around the server process.

A server is expected to always be available, whereas clients may come and go.[1] It is thus necessary that the server has a fixed and well-known address which can be reached by a client at any time and from anywhere.[1] The server will wait for a client to initiate contact and then perform various tasks requested by the client.[3] It is expected that a server may be performing tasks for multiple clients at any given time.

Notably, there is no direct communication between client processes in the client-server model.[3] All data-exchanges pass through the server, whether the task at hand involves only the client and server, or is a (nominal) interaction between clients. So, again, we see that the server lies at the center of the distributed application.

Since the server's address is fixed and known to all processes within the distributed application, and since all exchanges pass through the server, it is an advantage of this model that neither servers nor clients need to know the address of any client in advance of a communication session. If, for example, a client process is run from a mobile device such as a laptop or phone, the address of the client is expected to change, thus creating issues for any architecture that requires the client's address be known prior to data exchange.

One distinct disadvantage of the client-server model is that if traffic is intense, the server can become bogged down by servicing large volumes of requests, potentially leading to buffers overflowing, dropped data, or blocked requests. The available solution is to distribute the load among multiple servers or data-centers.[1] Unfortunately, obtaining this infrastructure can be cost-prohibitive.

Let's enumerate the key features of the client-server architecture (these will be put to use when deciding which architecture to use for *TerminalTalk*):

5. The server is always available to service requests.
6. The server has a fixed and well-known address.
7. The server does not need to know client addresses in advance of communication.
8. Clients do not directly communicate with one another.
9. Traffic intensive applications can be costly.

2.1.2 Peer-To-Peer Architecture

Unlike the client-server model, *peer-to-peer architecture* is inherently decentralized. Rather than a reliance on server processes peer-to-peer architecture "... exploits direct communication between pairs of intermittently connected hosts, called peers".[1] In fact, there generally are a minimal number (or even no) server processes utilized by the peer-to-peer application.

Under the peer-to-peer model, the resources required to perform application wide functions are divided amongst peers which "... are equally privileged, equipotent participants in the application".[4] Of course, this means that the available processing power and availability of data is determined entirely by the number and quality of peer connections at a given time. For example, if only a small number of peers are participating in an interprocess service, or their connections are unreliable, one can expect poor performance from the peer-to-peer application. On the other hand, since there is no theoretical limit to

the number of peers that can participate, there is the potential for the quality of service to grow without bound, as more and more peers add share their resources.

The latter result shows the potential of a peer-to-peer architecture to be particularly well suited for traffic-intensive applications. The application is essentially self-scalable, and so no changes or additions to the application code itself are required to grown and improve the service.[1] Furthermore, there are no prohibitive cost requirements for the addition of servers and data-centers. This is, perhaps, peer-to-peer architecture's greatest advantage over the client-server model.

Here's an enumeration of the key features of the peer-to-peer model:

10. Clients directly communicate.
11. The application is self-scalable and thereby well suited to traffic intensive applications.
12. There is no guarantee of service, as connections may be only intermittent.
13. There are minimal or no costs involved in obtaining and maintaining servers/data-centers.

2.1.3 TerminalTalk Architecture

The most salient feature of the *TerminalTalk* application relevant here is that the chat room is always expected to be available (see item 1). The most straight forward way to implement this is with an always available server. The client-server model, then, seems like a suitable choice (see item 5).

Perhaps a chat room could be "simulated" with a peer-to-peer processes that constantly tracks and updates currently connected user addresses. Though feasible, such a "simulated" chat room would be unnecessarily complicated when the client-server model can track the necessary addresses simply and easily (see item 6). Furthermore, there remains the necessity of providing a chat room when no peers are connected (item 12). So, a peer-to-peer architecture would not eliminate the need for a server. So, the client-to-server model still appears favorable.

If *TerminalTalk* were expanded to include direct two-way communication between users, perhaps then the peer-to-peer model would have advantages. For it seems unnecessary to burden the server with relaying messages between two users when the users could simply exchange addresses and communicate directly (item 10). With both direct two-way communication and public chat rooms available, it seems that features of both peer-to-peer and client-server architectures are desirable. It's unsurprising, then, that many chat applications utilize a hybrid of the client-server and peer-to-peer architectures.[5]

It seems improbable that a terminal based chat application such as *TerminalTalk* is going to get a large volume of use, given the multitudes of user-friendly chat applications available. So, items 11, 9, and 13 are non-factors. Thus, the client-server model emerges as the best application architecture for *TerminalTalk* .

2.2 Transport Layer Services

The network application developer must know what she can expect from the transport layer upon which her application depends. In the discussion that follows two prominent transport layer protocols will be examined, UDP and TCP. From the perspective of the application, however, the transport layer is essentially a black-box, providing services to the application by means unknown. Thus, the transport protocols will be examined in terms of the services they provide, rather than the mechanics by which they operate. The specific mechanisms of the transport layer, though interesting, are irrelevant to the task at hand.

2.2.1 TCP

The Transport Layer Protocol (TCP) provides two major services to the application layer—reliable data transfer and a connection-oriented service:

Reliable data transfer implies three things. First, the sending process can pass its application layer message to the transport layer with the certainty that the receiving process will obtain the message. Second, the received message will be error free. Third, it is guaranteed that messages will be received in exactly the same order in which they were sent.[6] Reliable data transfer greatly simplifies matters for the application designer as she does not need to concern herself with error checking / correcting or with contingencies such as out of sequence messages.

Before application layer messages are transfered with a *connection-oriented service*, "control" data are exchanged by the transport layers belonging to each end system in the connection. The purpose of this preliminary exchange is to establish the parameters of the connection. This initial exchange is often called *handshaking*. From the application's perspective, the important features of a connection-oriented service are that communication is full-duplex (simultaneous two-way), and that the connection is maintained until either the sending or receiving process initiates a disconnection processes.[1] Furthermore, as will be contrasted with UDP, the above described "handshaking" ensures that the receiving process exists and is available to exchange messages.

TCP synopsis:

14. Guaranteed delivery.
15. Error free delivery.
16. In order delivery.
17. Full-duplex communication.
18. Connection maintained until completed.
19. Checks that connection is available / exists.

2.2.2 UDP

The User Datagram Protocol (UDP) is a minimalist transport layer mechanism.[7] It is perhaps best described by the which services it lacks. For example, in contrast to TCP, UDP is often described as a *connectionless* protocol, for it makes no effort to establish a connection with the receiver or to negotiate the parameters of future data transmission.

> ...[UDP] does just about as little as a transport protocol can do....UDP takes messages from the application process, attaches source and destination port number fields... and passes the resulting segment to the network layer. [1]

With the exception of some minimal error checking, UDP does not make any guarantees at reliable data transfer. Segments may or may not arrive at the intended receiver, corrupted or not, and in no particular order.

This may lead to the impression that use of UDP would be generally undesirable. However, there are advantages to the protocol. Kurose and Ross mention these advantages (my comments are in parentheses):[1]

- Finer application-level control over what/when data is sent.
- No connection establishment (i.e., its faster).

- No connection state (i.e., less processor / memory intensive).
- Small packet overhead (8 bytes to TCP's 20).

More specific to the application layer, if it is not essential that all data is delivered in tact, as is the case in applications such as video transmission, where pixels or frames may be dropped without noticeable effect, then the time saved by skipping error correction and handshaking may be worth the cost of occasionally losing data. Likewise, if the application is going to generate many processes, or resource intensive processes, then UDP may help minimize the burden carried by the end-system.

UDP summary :

20. Connectionless.
21. Unreliable.
22. Avoids time delays required by reliable, connection-oriented protocols.
23. Less processor / memory intensive.

2.2.3 Terminal Talk Transport Layer

The additional delays incurred by reliable data transfer and connection establishment are generally less than a second. For real time applications, such as online gaming or internet telephony, this delay is significant. However, for a chat application it seems that a message could be delayed up to a few dozen seconds without adverse effects. Thus, item 22 is not a significant factor for *TerminalTalk* .

Unreliable data transfer, on the other hand, has the potential to cause more serious issues. Consider for example, warning your friend to apply Tyrannosaurus repellent before heading to the supper-market, as one such Cretaceous beast recently escaped your top-secret laboratory. Should this message fail to reach your friend, the consequences could be fatal. Thus, item 14 is a desirable for *TerminalTalk* .

In a similar vein, imagine if "Tyrannosaurus repellent" were jumbled so as to read "Aunt Sour's Yarn repellent" in the received message. Unfortunately, yarn repellent just wouldn't do against the dangerous dino. So item 15 must be added to the *TerminalTalk* list of requirements.

The order in which messages are received is also important. Clearly, in any conversation coherence requires an orderly sequence of thoughts. More specific to *TerminalTalk* however, imagine what would happen if a command where received by the server out of order. If, for example, the `\disconnect` command were received before one or more of the user's messages, none of those messages would make it through to the chat room. This is an undesirable result, and so item 16 is added.

In section 1.2 it was stated that *TerminalTalk* would announce both that a user had entered and exited the chat room. To do this it is necessary that the server keep track of the user's connection. Relying on the `\disconnect` command won't do as the user may otherwise become disconnected (and it's quite embarrassing to continue talking to someone who has left earshot without your knowing). UPD's connectionless protocol, then, is inadequate for *TerminalTalk* .

So TCP is the best option for *TerminalTalk* . This conclusion is unsurprising, as most text-centric applications utilize TCP:

> For many applications—such as electronic mail, file transfer, remote host access, Web documents transfers, and financial applications—data loss can have devastating consequences.... [1]

2.3 Application Layer Protocol

Effective communication requires a set of rules or guidelines that give information exchanges structure. For example, in a conversation it is expected that interlocutors do not speak simultaneously. Doing so would be result in neither participant being understood. Likewise, it is expected that words are spoken at an audible volume, and at an intelligible but efficient pace.

The need for structured communication is all the more essential in exchanges between computers. Unlike people, computers generally are not adaptable, and thus, the exact structure of a "conversation" between computers must be precisely defined. Sets of precise definitions of the structure of information exchanges are known as communication protocols:

> To communicate, the computers must have a common language, and they must follow rules so that both the client and the server know what to expect. The language and rules of communication are defined in a communications protocol. All client-server protocols operate in the application layer. The application-layer protocol defines the basic patterns of the dialogue. [3]

There are many application layer protocols in existence. However, since *TerminalTalk* has rather minimal needs it will have its own bare-bones protocol optimized for its purposes. Below is a functional description of the protocol:

When a user first connects, there will be an initial handshaking phase similar to TCP's "three-way handshake".[1] First, the client process will request a connection with the server and wait for a response. Upon receiving the request, the server will allocate a new socket for the connection, request a moniker from the client process, and then wait for a response. Finally, upon receiving the server's moniker request, the client process will send the moniker and the connection is considered established.

After the handshaking phase, the server will check each received message to see whether it is a command. If the first character of a message begins is '\' it is considered a command, and is treated accordingly. Any message which does not begin with '\' is considered a 'missive' (a message intended to be seen by other connected users) and is treated accordingly.

And that's it.

3 Implementation

The following is a functional overview of the code implementation of *TerminalTalk* . The innards of employed functions will not be elucidated here, however part II of this paper includes a complete API of the code (and the source code itself is available here).

It is worth noting that in the early development of *TerminalTalk* this tutorial was used as an initial template.[8] However, as will be seen, *TerminalTalk* has evolved a great deal since then.

3.1 Client Process

Users can start *TerminalTalk* by opening their terminals and entering:

```
python TerminalTalk.py [moniker]
```

If a user does not include the optional moniker when starting *TerminalTalk* the application assigns her "Anonymous" as a temporary moniker (until she uses the `\setmoniker` command).

```
if( len(sys.argv) < 2 ):
  moniker = "Anonymous"
else:
  moniker = sys.argv[1]
```

The client then proceeds with the "three-way handshake" described in section 2.3.

```
# Create socket
megaphone = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Three-way Handshake
try:
  megaphone.connect(server_address)

  # Wait for server to request moniker
  request = megaphone.recv(buffer_size)
  if request == "moniker":
    megaphone.send(moniker)
    # Connection established.

except:
  # If the client failed to connect with server, inform user and exit.
  print('Could not connect to TerminalTalk servers. Please try again later.')
  sys.exit()
```

The client must listen for two events: (1) a message inputed by the user, and (2) a message sent by the server. To do this, the `select` module is employed.[9] The `select.select()` returns a list of all interfaces ready to be read in the user's end system. By looping through this list, and checking for `stdin` and the `megaphone` socket, the program intercepts all incoming messages and takes the appropriate actions:

```
# Create a list to keep track of connections (terminal input and megaphone)
connections = [megaphone, sys.stdin]

eavesdrop() # Prompt user for input
while True:
  # Get a list of readable sockets / inputs
  readables, writables, errors = select.select(connections, [], [])

  for telegraph_i in readables:
    # See if a message has been sent from server
    if telegraph_i == megaphone:
      missive = telegraph_i.recv(buffer_size)
      if missive:
        sys.stdout.write(missive)
        eavesdrop() # Prompt user for input
      else:
        # If misssive returns False, the connection is broken.
        print("\n The connection has been lost... please reconnect.")
        sys.exit()
    else:
      # See if the user has entered a message
      verbiage = sys.stdin.readline()
      megaphone.send(verbiage)
      eavesdrop() # Prompt user for input
```

3.2 Server Process

The *TerminalTalk* server process has a specialized socket for receiving new connections (`ear_trumpet`). When a user requests a new connection the server allocates a new socket and adds it to a list of `connections`. It also instantiates an `Orator` object which tracks the connected user's information. The new `Orator` is kept in the `orators` list.

```
# Create the server socket
ear_trumpet = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ear_trumpet.bind( server_address )
ear_trumpet.listen(5)

connections = [ear_trumpet] # A list that keeps track of active sockets
orators = [] # A list of active Orators
```

As with the client process, the `select.select()` function is used to loop through readable interfaces. If the `ear_trumpet` socket is ready to be read, this means that a new user is requesting to connect. In this instance the server process plays its role in the "three-way handshake" and allocates the afformentioned socket and `Orator`. The server then announces the user's arrival to the chat room.

```
while True:
  # Use select function to check which sockets are ready to be read.
  readables, writables, errors = select.select(connections, [], [])

  # Loop through readables. Address each.
  for telegraph_i in readables:
    # A new connection has been requested
    if telegraph_i == ear_trumpet:
      # Accept connection
      file_descriptor, address = telegraph_i.accept()
      connections.append(file_descriptor)

      # Add an Orator object to orators to keep track of user data
      orators.append( Orator(file_descriptor) )
      file_descriptor.send("moniker")                    # Request moniker
      moniker = file_descriptor.recv(buffer_size)        # Wait to get a response
      orators[len(orators)-1].moniker = moniker # Assign moniker to Orator


      # State that a new user has entered.
      entrance_message = moniker + " has entered."
      megaphone( entrance_message, connections, ear_trumpet, orators )
      print(entrance_message)
```

If a user socket is ready to be read, the server checks whether the received message is a command or a missive and respond accordingly. If the attempt to read from the socket fails, the program infers that the connection has been lost. The server will first announce to the chat room that the user has exited, and then removed the user's socket and `Orator`.

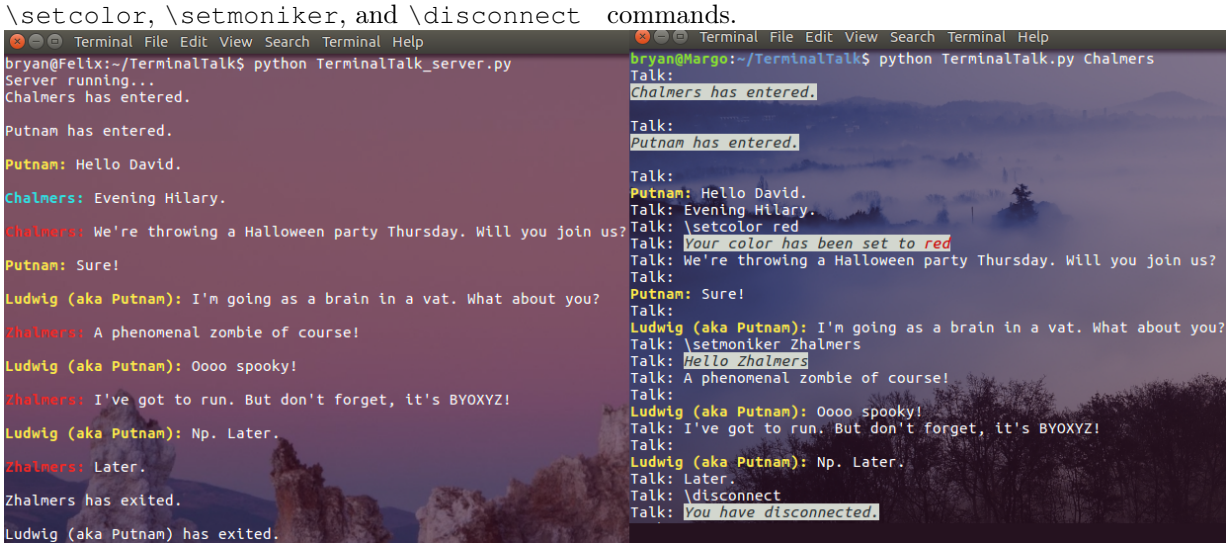
```
# Now read the orator socket
try:
  verbiage = telegraph_i.recv(buffer_size)
  if verbiage:
    # Check whether the recieved missive is a command
    if verbiage[0] == "\\":
      # Decode and action command
      obey(verbiage, orators[orator_index], telegraph_i)
    else:
      pontificate( orators[orator_index], verbiage, connections, ear_trumpet, orators )
      print( font.bold( orators[orator_index].color + orators[orator_index].moniker + ":
    " + font.Styles.RESET ) + verbiage)
except:
  # This indicates that the connection has been broken.
  missive = orators[orator_index].moniker + " has exited."
  pontificate( orators[orator_index], missive, connections, ear_trumpet, orators )
  print(missive)

  # Remove from connections and orators list
```

```
19      telegraph_i.close()
        orators.remove( orators[orator_index] )
21      connections.remove( telegraph_i )
        continue
```

3.3 Demonstration

Beneath is a screen shot of *TerminalTalk* in action. On the left hand side of the image is the server process. It displays all of the activity taking place in the chat room. On the right hand side is the client side of *TerminalTalk* as seen by the user. In addition to conversation, the example shows use of the `\setcolor`, `\setmoniker`, and `\disconnect` commands.



Part II

API

4 Namespace Index

4.1 Packages

Here are the packages with brief descriptions (if available):

5 Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

6 File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

7 Namespace Documentation

7.1 client Namespace Reference

Functions and definitions specific to TerminalTalk client.

7.1.1 Detailed Description

Functions and definitions specific to TerminalTalk client.

7.2 font Namespace Reference

A collection of functions that style fonts in terminal.

7.2.1 Detailed Description

A collection of functions that style fonts in terminal.

Each function flanks the inputed text with ANSI codes.

7.3 server Namespace Reference

Functions and definitions specific to TerminalTalk server.

7.3.1 Detailed Description

Functions and definitions specific to TerminalTalk server.

8 Class Documentation

8.1 font.constants.Colors Class Reference

Contains constants for ANSI color codes.

Static Public Attributes

- string **BLACK** = '\033[30m]'
- string **RED** = '\033[31m'
- string **GREEN** = '\033[32m'
- string **YELLOW** = '\033[33m'
- string **BLUE** = '\033[34m'
- string **MAGENTA** = '\033[35m'
- string **CYAN** = '\033[36m'
- string **WHITE** = '\033[37m'

8.2 Detailed Description

Contains constants for ANSI color codes.

Prefix a string with color constant to change subsequent text to the color. Example: print(font.Colors.↩ RED + "This text will be red.")

The documentation for this class was generated from the following file:

- /TerminalTalk/font/constants.py

8.3 server.Orator.Orator Class Reference

Stores information about a connected client.

Public Member Functions

- def __init__ (self, telegraph)
  Constructor for Orator class.

Public Attributes

- **telegraph**
- **moniker**
- **color**

8.3.1 Detailed Description

Stores information about a connected client.

An Orator objects is used to represent a client. Each client has a moniker and color associated with them. The moniker is provided by the client. The color is assigned randomly each time the client connects.

8.3.2 Constructor & Destructor Documentation

8.3.2.1 def server.Orator.Orator.__init__ (self, telegraph)

Constructor for Orator class.

Parameters

telegraph	The socket used to connect with client.

The documentation for this class was generated from the following file:

- /TerminalTalk/server/Orator.py

8.4 font.constants.Styles Class Reference

Contains constants for ANSI text style codes.

Static Public Attributes

- string **RESET** = '\033[0m'
- string **BOLD** = '\033[1m'
- string **BOLD_OFF** = '\033[22m'
- string **ITALIC** = '\033[3m'

- string **ITALIC_OFF** = '\033[23m'
- string **UNDERLINE** = '\033[4m'
- string **UNDERLINE_OFF** = '\033[24m'
- string **STRIKETHROUGH** = '\033[9m'
- string **STRIKETHROUHG_OFF** = '\033[29m'
- string **INVERSE** = '\033[7m'
- string **INVERSE_OFF** = '\033[27m'

8.4.1 Detailed Description

Contains constants for ANSI text style codes.

Prefix a string with a style constant to change subsequent text style. Example: print(font.Styles.BOLD + "This text will be bold.")

The documentation for this class was generated from the following file:

- /TerminalTalk/font/constants.py

9 File Documentation

9.1 /TerminalTalk/client/eavesdrop.py File Reference

Functions

- def client.eavesdrop.eavesdrop ()
  Monitors for user input from terminal.

9.2 /TerminalTalk/font/constants.py File Reference

Defines classes contains ANSI constants for formating text.

Classes

- class font.constants.Styles
  Contains constants for ANSI text style codes.
- class font.constants.Colors
  Contains constants for ANSI color codes.

9.2.1 Detailed Description

Defines classes contains ANSI constants for formating text.

9.3 /TerminalTalk/font/functions.py File Reference

Defines functions for surronding strings with ANSI codes for formating text.

Functions

- def font.functions.get_color_code (color)
  Returns ANSI color code for specified color.
- def font.functions.blue (txt)
  Makes inputed string blue.
- def font.functions.cyan (txt)
  Makes inputed string cyan.
- def font.functions.green (txt)
  Makes inputed string green.
- def font.functions.magenta (txt)
  Makes inputed string magenta.
- def font.functions.red (txt)
  Makes inputed string red.
- def font.functions.yellow (txt)
  Makes inputed string red.
- def font.functions.underline (txt)
  Makes inputed string underline.
- def font.functions.bold (txt)
  Makes inputed string bold.
- def font.functions.italic (txt)
  Makes inputed string italic.
- def font.functions.default (txt)
  Gives inputed string default formatting.
- def font.functions.highlight (txt)
  Highlights inputed string.

9.3.1 Detailed Description

Defines functions for surronding strings with ANSI codes for formating text.

9.3.2 Function Documentation

9.3.2.1 def font.functions.blue (txt)

Makes inputed string blue.

Returns with inputed string with the prefix '\033[34m' and suffix '\033[0m'. The prefix is the ANSI code for blue foreground. The suffix is the ANSI code for reset (white foreground, black background).

Parameters

txt	A string. Will be displayed as blue.

9.3.2.2 def font.functions.bold (txt)

Makes inputed string bold.

Returns with inputed string with the prefix '\033[1m' and suffix '\033[22m'. The prefix is the ANSI code for bold font on. The suffix is the ANSI code for bold font off.

Parameters

txt	A string. Will be made bold.

9.3.2.3 def font.functions.cyan (txt)

Makes inputed string cyan.

Returns with inputed string with the prefix '\033[36m' and suffix '\033[0m'. The prefix is the ANSI code for cyan foreground. The suffix is the ANSI code for reset (white forground, black background).

Parameters

txt	A string. Will be displayed as cyan.

9.3.2.4 def font.functions.default (txt)

Gives inputed string default formatting.

Returns with inputed string with the prefix '\033[0m'. The prefix is the ANSI code for reset (white foreground, black background).

Parameters

txt	A string. Will be formatted as default.

9.3.2.5 def font.functions.green (txt)

Makes inputed string green.

Returns with inputed string with the prefix '\033[32m' and suffix '\033[0m'. The prefix is the ANSI code for green foreground. The suffix is the ANSI code for reset (white forground, black background).

Parameters

txt	A string. Will be displayed as green.

9.3.2.6 def font.functions.highlight (txt)

Highlights inputed string.

Returns with inputed string with the prefixes '\033[47m', '\033[30m', and'\033[3m', and with the suffix '\033[0m'. The prefix is the ANSI code for italic, white background, and black foreground. The suffix is the ANSI code for reset (white forground, black background).

Parameters

txt	A string. Will be highlighted.

9.3.2.7 def font.functions.italic (txt)

Makes inputed string italic.

Returns with inputed string with the prefix '\033[3m' and suffix '\033[23m'. The prefix is the ANSI code for italic on. The suffix is the ANSI code for italic off.

Parameters

txt	A string. Will be displayed as italic.

9.3.2.8 def font.functions.magenta (txt)

Makes inputed string magenta.

Returns with inputed string with the prefix '\033[35m' and suffix '\033[0m'. The prefix is the ANSI code for magenta foreground. The suffix is the ANSI code for reset (white forground, black background).

Parameters

txt	A string. Will be displayed as magenta.

9.3.2.9 def font.functions.red (txt)

Makes inputed string red.

Returns with inputed string with the prefix '\033[31m' and suffix '\033[0m'. The prefix is the ANSI code for red foreground. The suffix is the ANSI code for reset (white forground, black background).

Parameters

txt	A string. Will be displayed as red.

9.3.2.10 def font.functions.underline (txt)

Makes inputed string underline.

Returns with inputed string with the prefix '\033[4m' and suffix '\033[24m'. The prefix is the ANSI code for underline on. The suffix is the ANSI code for underline off.

Parameters

txt	A string. Will be underlined.

9.3.2.11 def font.functions.yellow (txt)

Makes inputed string red.

Returns with inputed string with the prefix '\033[33m' and suffix '\033[0m'. The prefix is the ANSI code for red foreground. The suffix is the ANSI code for reset (white forground, black background).

Parameters

txt	A string. Will be displayed as red.

10 TerminalTalk/server/obey.py File Reference

Defines functions for procesing user inputed commands.

Functions

- def server.obey.setcolor (telegraph, orator, color)
  Changes user's display color to specified value.
- def server.obey.setmoniker (telegraph, orator, moniker)
  Changes user's moniker to specifed value.
- def server.obey.disconnect (telegraph, orator, unsused_string)
  Disconnects user from TerminalTalk chatroom.
- def server.obey.obey (full_command, orator, telegraph)
  Processes and executes user specified commands.

10.1 Detailed Description

Defines functions for procesing user inputed commands.

10.2 Function Documentation

10.2.1 def server.obey.disconnect (telegraph, orator, unsused_string)

Disconnects user from TerminalTalk chatroom.

Parameters

telegraph	The user's socket.
orator	The Orator object tracking the user's info.
unsused_string	A string. Unused. Can contain any value.

10.2.2 def server.obey.obey (full_command, orator, telegraph)

Processes and executes user specified commands.

The obey function parses the full_command parameter into command and argument strings. Using the command string is finds and calls the appropriate function (setcolor, setmoniker, disconnect) and passes the argument string and any other necessary parameters to it.

Parameters

full_command	The string containting the command and argument sent by user.
orator	The Orator object tracking the user's info
telegraph	The user's socket.

10.2.3 def server.obey.setcolor (telegraph, orator, color)

Changes user's display color to specified value.

This function is inteded to be called by the obey function. When called, the function calls the get_color_code functions from the font module. If the color is available, setcolor changes the color attribute of the specified Orator object and confirms the change through the specifed socket (telegraph). If the color is unavailable, the user is informed through the specified socket.

Parameters

telegraph	The user's socket.
orator	The Orator object tracking the user's info.
color	A string specifying a color. Example: "green"

10.2.4 def server.obey.setmoniker (telegraph, orator, moniker)

Changes user's moniker to specifed value.

This function is inteded to be called by the obey function. When called, the function sets the moniker attribute of the specified Orator object to the moniker parameter.

Parameters

telegraph	The user's socket.
orator	The Orator object tracking the user's info.
moniker	A string containing the new moniker.

10.3 /TerminalTalk/server/Orator.py File Reference

Classes

- class server.Orator.Orator

  Stores information about a connected client.

10.4 /TerminalTalk/server/pontification.py File Reference

Functions

- def server.pontification.megaphone (verbiage, connections, ear_trumpet, orators)

  Make announcements from server to everyone connected.
- def server.pontification.pontificate (orator, verbiage, connections, ear_trumpet, orators)

  Transmit a clients message to all other connected users.

10.4.1 Function Documentation

10.4.1.1 def server.pontification.megaphone (verbiage, connections, ear_trumpet, orators)

Make announcements from server to everyone connected.

The message will be formatted (see font.highlight) so as to stick out from all other text. Every connected user will see the message.

Parameters

verbiage	The message displayed.
connections	A list of all current connections (sockets).
ear_trumpet	The server's socket. Used to ensure message isn't sent to server itself (this would break pipe).
orators	A list of Orator objects for currently connected clients. If a disconnection is discovered while function is being run, it is necessary to remove the object from the list.

10.4.1.2 def server.pontification.pontificate (orator, verbiage, connections, ear_trumpet, orators)

Transmit a clients message to all other connected users.

Parameters

orator	The sending client's Orator object.
verbiage	The message displayed.
connections	A list of all current connections (sockets).
ear_trumpet	The server's socket. Used to ensure message isn't sent to server itself (this would break pipe).
orators	A list of Orator objects for currently connected clients. If a disconnection is discovered while function is being run, it is necessary to remove the object from the list.

10.5 /TerminalTalk/TerminalTalk.py File Reference

Main code for TerminalTalk client.

Variables

- string **TerminalTalk.moniker** = "Anonymous"
- int **TerminalTalk.buffer_size** = 2
- int **TerminalTalk.server_port** = 7777
- string **TerminalTalk.server_ip** = "192.168.1.77"
- tuple **TerminalTalk.server_address** = (server_ip, server_port)
- **TerminalTalk.megaphone** = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
- **TerminalTalk.request** = megaphone.recv(buffer_size)

- list **TerminalTalk.connections** = [megaphone, sys.stdin]
- **TerminalTalk.readables**
- **TerminalTalk.writables**
- **TerminalTalk.errors**
- **TerminalTalk.missive** = telegraph_i.recv(buffer_size)
- **TerminalTalk.verbiage** = sys.stdin.readline()

10.5.1 Detailed Description

Main code for TerminalTalk client.

Run this file to use as client.

10.6 /TerminalTalk/TerminalTalk_server.py File Reference

Main code for TerminalTalk server.

Variables

- int **TerminalTalk_server.port** = 7777
- int **TerminalTalk_server.buffer_size** = 2
- tuple **TerminalTalk_server.server_address** = ("", port)
- list **TerminalTalk_server.connections** = [ ]
- list **TerminalTalk_server.orators** = [ ]
- **TerminalTalk_server.ear_trumpet** = socket.socket(socket.AF_INET, socket.SOCK_STR↩ EAM)
- **TerminalTalk_server.readables**
- **TerminalTalk_server.writables**
- **TerminalTalk_server.errors**
- **TerminalTalk_server.file_descriptor**
- **TerminalTalk_server.address**
- **TerminalTalk_server.moniker** = file_descriptor.recv(buffer_size)
- string **TerminalTalk_server.entrance_message** = moniker+" has entered."
- int **TerminalTalk_server.orator_index** = 0
- **TerminalTalk_server.verbiage** = telegraph_i.recv(buffer_size)
- string **TerminalTalk_server.missive** = orators[orator_index].moniker+" has exited."

10.6.1 Detailed Description

Main code for TerminalTalk server.

Run this file to start server.

References

[1] J. F. Kurose and K. W. Ross, Computer networking: a top-down approach. Boston: Pearson, 2017. 2, 3, 4, 5, 6, 7

[2] "Application Layer". En.wikipedia.org. N.p., 2016. Web. 5 Nov. 2016. 2

[3] "Client–Server Model". En.wikipedia.org. N.p., 2016. Web. 5 Nov. 2016. 3, 7

[4] "Peer-To-Peer". En.wikipedia.org. N.p., 2016. Web. 6 Nov. 2016. 3

[5] "Instant Messaging". En.wikipedia.org. N.p., 2016. Web. 6 Nov. 2016. 4

[6] "Transmission Control Protocol". En.wikipedia.org. N.p., 2016. Web. 12 Nov. 2016. 5

[7] "User Datagram Protocol". En.wikipedia.org. N.p., 2016. Web. 12 Nov. 2016. 5

[8] "Python Socket – Chat Server And Client With Code Example". BinaryTides. N.p., 2016. Web. 5 Nov. 2016. 7

[9] "16.1. Select — Waiting For I/O Completion — Python 2.7.12 Documentation". Docs.python.org. N.p., 2016. Web. 5 Nov. 2016. 8