pyVoIP

Release 2.0.0a5

Tayler J Porter

Feb 14, 2024

CONTENTS:

PyVoIP is a pure python VoIP/SIP/RTP library. Currently, it supports PCMA, PCMU, and telephone-event.

Please note this is is still in development and can only originate calls with PCMU. In future, it will be able to initiate calls in PCMA as well.

This library does not depend on a sound library, i.e. you can use any sound library that can handle linear sound data such as pyaudio or even wave. Keep in mind PCMU only supports 8000Hz, 1 channel, 8 bit audio.

In this documentation we will use the following terms:

client

For the purposes of this documentation, the term *client* will be defined as the person calling this library.

user

For the purposes of this documentation, the term *user* will be defined as the programmer, i.e. the 'server-side' if using the Client-Server model.

CHAPTER

ONE

EXAMPLES

Here we will go over a few basic phone setups.

1.1 Setup

PyVoIP uses a *VoIPPhone* class to receive and initiate phone calls. The settings for our phone are passed via the *VoIPPhoneParameter* dataclass. When a call is received, a new instance of a *VoIPCall* is initialized. You can overwrite this class in initialization of VoIPPhone.

In this example, we are importing *CredentialsManager*, *VoIPPhone*, *VoIPPhoneParameter*, *VoIPCall*, and *InvalidStateError*. *CredentialsManager* stores and retreives passwords for authentication with registrars. *VoIPPhone* is the main class for our softphone. *VoIPPhoneParameter* is the settings for our *VoIPPhone*. *VoIPCall* will be used to create our custom answering class. An *InvalidStateError* is thrown when you try to perform an impossible command. For example, denying the call when the phone is already answered, answering when it's already answered, etc.

The following will create a phone that answers and automatically hangs up:

```
from pyVoIP.credentials import CredentialsManager
from pyVoIP.VoIP.call import VoIPCall
from pyVoIP.VoIP.error import InvalidStateError
from pyVoIP.VoIP.phone import VoIPPhone, VoIPPhoneParamter

class Call(VoIPCall):

    def ringing(self, invite_request):
        try:
            self.answer()
            self.hangup()
        except InvalidStateError:
            pass

if __name__ == "__main__":
    cm = CredentialsManager()
    cm.add(<SIP server username>, <SIP server password>)
    params = VoIPPhoneParamter(<SIP server IP>, <SIP server port>, <SIP server user>, cm,
↪ bind_ip=<Your computers local IP>, call_class=Call)
    phone = VoIPPhone(params)
    phone.start()
    input('Press enter to disable the phone')
    phone.stop()
```

1.2 Announcement Board

Let's say you want to make a phone that when you call it, it plays an announcement message, then hangs up. We can accomplish this with the builtin libraries wave, audioop, time, and by importing *CallState*.

```python
from pyVoIP.credentials import CredentialsManager
from pyVoIP.VoIP.call import VoIPCall
from pyVoIP.VoIP.error import InvalidStateError
from pyVoIP.VoIP.phone import VoIPPhone, VoIPPhoneParamter
import time
import wave

class Call(VoIPCall):

    def ringing(self, invite_request):
        try:
            f = wave.open('announcment.wav', 'rb')
            frames = f.getnframes()
            data = f.readframes(frames)
            f.close()

            call.answer()
            call.write_audio(data)  # This writes the audio data to the transmit buffer,␣
↪this must be bytes.

            stop = time.time() + (frames / 8000)  # frames/8000 is the length of the␣
↪audio in seconds. 8000 is the hertz of PCMU.

            while time.time() <= stop and call.state == CallState.ANSWERED:
                time.sleep(0.1)
            call.hangup()
        except InvalidStateError:
            pass
        except:
            call.hangup()

if __name__ == "__main__":
    cm = CredentialsManager()
    cm.add(<SIP server username>, <SIP server password>)
    params = VoIPPhoneParamter(<SIP server IP>, <SIP server port>, <SIP server user>, cm,
↪ bind_ip=<Your computer's local IP>, call_class=Call)
    phone = VoIPPhone(params)
    phone.start()
    input('Press enter to disable the phone')
    phone.stop()
```

Something important to note is our wait function. We are currently using:

```python
stop = time.time() + (frames / 8000)  # The number of frames/8000 is the length of the␣
↪audio in seconds.

while time.time() <= stop and call.state == CallState.ANSWERED:
    time.sleep(0.1)
```

This could be replaced with `time.sleep(frames / 8000)`. However, doing so will not cause the thread to automatically close if the user hangs up, or if `VoIPPhone().stop()` is called. Using the while loop method will fix this issue. The `time.sleep(0.1)` inside the while loop is also important. Supplementing `time.sleep(0.1)` for `pass` will cause your CPU to ramp up while running the loop, making the RTP (audio being sent out and received) lag. This can make the voice audibly slow or choppy.

> **Important Note:** *Audio must be 8 bit, 8000Hz, and Mono/1 channel. You can accomplish this in a free program called Audacity. To make an audio recording Mono, go to Tracks > Mix > Mix Stereo Down to Mono. To make an audio recording 8000 Hz, go to Tracks > Resample… and select 8000, then ensure that your 'Project Rate' in the bottom left is also set to 8000. To make an audio recording 8 bit, go to File > Export > Export as WAV, then change 'Save as type:' to 'Other uncompressed files', then set 'Header:' to 'WAV (Microsoft)', then set the 'Encoding:' to 'Unsigned 8-bit PCM'*

1.3 IVR/Phone Menus

We can use the following code to create IVR Menus. Currently, we cannot make 'breaking' IVR menus. Breaking IVR menus in this context means, a user selecting an option mid-prompt will cancel the prompt, and start the next action. Support for breaking IVR's will be made in the future. For now, here is the code for a non-breaking IVR:

```python
from pyVoIP.credentials import CredentialsManager
from pyVoIP.VoIP.call import VoIPCall
from pyVoIP.VoIP.error import InvalidStateError
from pyVoIP.VoIP.phone import VoIPPhone, VoIPPhoneParamter
import time
import wave

class Call(VoIPCall):

    def ringing(self, invite_request):
        try:
            f = wave.open('prompt.wav', 'rb')
            frames = f.getnframes()
            data = f.readframes(frames)
            f.close()

            call.answer()
            call.write_audio(data)

            while call.state == CallState.ANSWERED:
                dtmf = call.get_dtmf()
                if dtmf == "1":
                    if call.transfer("sales")  # Transfer to same registrar
                      return
                elif dtmf == "2":
                    if call.transfer(uri="<100@different_regisrar.com>")
                      return
                time.sleep(0.1)
        except InvalidStateError:
            pass
        except:
            call.hangup()
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    cm = CredentialsManager()
    cm.add(<SIP server username>, <SIP server password>)
    params = VoIPPhoneParamter(<SIP server IP>, <SIP server port>, <SIP server user>, cm,
↪ bind_ip=<Your computer's local IP>, call_class=Call)
    phone = VoIPPhone(params)
    phone.start()
    input('Press enter to disable the phone')
    phone.stop()
```

Please note that `get_dtmf()` is actually `get_dtmf(length=1)`, and as it is technically an `io.StringBuffer()`, it will return "" instead of `None`. This may be important if you wanted an 'if anything else, do that' clause. Lastly, VoIPCall stores all DTMF keys pressed since the call was established; meaning, users can press any key they want before the prompt even finishes, or may press a wrong key before the prompt even starts.

1.4 Call State Handling

We can use the following code to handle various states for calls:

```
from pyVoIP.credentials import CredentialsManager
from pyVoIP.VoIP.call import VoIPCall
from pyVoIP.VoIP.error import InvalidStateError
from pyVoIP.VoIP.phone import VoIPPhone, VoIPPhoneParamter
import time
import wave

class Call(VoIPCall):

    def progress(self, request):
        print('Progress')
        super().progress(request)

    def busy(self, request):
        print('Call ended - callee is busy')
        super().busy(request)

    def answered(self, request):
        print('Answered')
        super().answered()

    def bye(self):
        print('Bye')
        super().bye()

if __name__ == '__main__':
    cm = CredentialsManager()
    cm.add(<SIP server username>, <SIP server password>)
    params = VoIPPhoneParamter(<SIP server IP>, <SIP server port>, <SIP server user>, cm,
↪ bind_ip=<Your computer's local IP>, call_class=Call)
    phone = VoIPPhone(params)
    phone.start()
```

(continues on next page)

(continued from previous page)

```
    phone.call(<Phone Number>)
    input('Press enter to disable the phone\n')
    phone.stop()
```

CHAPTER

TWO

GLOBALS

2.1 Global Variables

There are a few global variables that may assist you if you're having problems with the library.

pyVoIP.DEBUG = False
If set to true, pyVoIP will print debug messages that may be useful if you need to troubleshoot or open a GitHub issue.

pyVoIP.TRANSMIT_DELAY_REDUCTION = 0.0
The higher this variable is, the more often RTP packets are sent. This *should* only ever need to be 0.0. However, when testing on Windows, there has sometimes been jittering, setting this to 0.75 fixed this in testing, but you may need to tinker with this number on a per-system basis.

pyVoIP.ALLOW_BASIC_AUTH = False
Controls whether Basic authentication (**RFC 7617**) is allowed for SIP authentication. Basic authentication is deprecated as it will send your password in plain-text, likely in the clear (unencrypted) as well. As such this is disabled be default.

pyVoIP.ALLOW_MD5_AUTH = True
MD5 Digest authentication is deprecated per RFC 8760 Section 3 as it a weak hash. However, it is still used often so it is enabled by default.

pyVoIP.REGISTER_FAILURE_THRESHOLD = 3
If registration fails this many times, VoIPPhone's status will be set to FAILED and the phone will stop.

pyVoIP.ALLOW_TLS_FALLBACK = False
If this is set to True TLS will fall back to TCP if the TLS handshake fails. This is off by default, as it would be irresponsible to have a security feature disabled by default.

This feature is currently not implemented.

pyVoIP.TLS_CHECK_HOSTNAME = True
Is used to create SSLContexts. See Python's documentation on check_hostname for more details.

You should use the *set_tls_security* function to change this variable.

pyVoIP.TLS_VERIFY_MODE = True
Is used to create SSLContexts. See Python's documentation on verify_mode for more details.

You should use the *set_tls_security* function to change this variable.

pyVoIP.SIP_STATE_DB_LOCATION = ":memory:"
This variable allows you to save the SIP message state database to a file instead of storing it in memory which is the default. This is useful for debugging, however pyVoIP does not delete the database afterwards which will cause an Exception upon restarting pyVoIP. For this reason, we recommend you do not change this variable in production.

2.2 Global Functions

pyVoIP. set_tls_security(verify_mode: VerifyMode) -> None

This method ensures that TLS_CHECK_HOSTNAME and TLS_VERIFY_MODE are set correctly depending on the TLS certificate verification settings you want to use.

CHAPTER

THREE

VOIP - THE BRIDGE BETWEEN SIP AND RTP

The VoIP module coordinates between the SIP and RTP modules in order to create an effective Voice over Internet Protocol system. The VoIP system is made for your convenience, and if you have a particularly intricate situation, you can override the SIP module on initialization to fit your use case.

3.1 Errors

There are three errors under `pyVoIP.VoIP.error`.

exception **pyVoIP.VoIP.error.InvalidStateError**

This is thrown by *VoIPCall* when you try to perform an action that cannot be performed during the current *CallState*. For example denying a call that has already been answered, hanging up a call that hasn't been answered yet, or has already ended.

exception **pyVoIP.VoIP.error.InvalidRangeError**

This is thrown by *VoIPPhone* when you define the RTP port ranges as rtp_port_low > rtp_port_high. However, this is not checked by *VoIPCall*, so if you are using your own class instead of VoIPPhone, make sure these ranges are correct.

exception **pyVoIP.VoIP.error.NoPortsAvailableError**

This is thrown when a call is attempting to be initiated but no RTP ports are available.

3.2 Enums

enum **pyVoIP.VoIP.call.CallState**

CallState is an Enum with six attributes.

CallState.DIALING

This CallState is used to describe when a *user* has originated a call to a *client*, but it has yet to be answered.

In this state, you can use `VoIPCall.cancel()`.

CallState.RINGING

This CallState is used to describe when a *client* is calling, but the call has yet to be answered.

In this state, you can use `VoIPCall.answer()` or `VoIPCall.deny()`.

CallState.PROGRESS

This CallState is used when a 183 Session Progress response is received on a call that is dialing.

In this state, you can use `VoIPCall.answer()`, `VoIPCall.deny()`, or `VoIPCall.cancel()`.

CallState.ANSWRED
This CallState is used to describe when a call has been answered and is active.

In this state, you can use `VoIPCall.hangup()`.

CallState.CANCELING
This CallState is used when a dialing call is canceled with `VoIPCall.cancel()`.

CallState.ENDED
This CallState is used to describe when a call has been terminated.

enum **pyVoIP.VoIP.status.PhoneStatus**
PhoneStatus is an Enum with five attributes.

PhoneStatus.INACTIVE
This PhoneStatus is used when `VoIPPhone.start()` has not been called, or after the phone has fully stopped after calling `VoIPPhone.stop()`.

PhoneStatus.REGISTERING
This PhoneStatus is used when `VoIPPhone.start()` has been called, but has not finished starting, or when the phone is re-registering.

PhoneStatus.REGISTERED
This PhoneStatus is used when `VoIPPhone` has finished starting successfully, and is ready for use.

PhoneStatus.DEREGISTERING
This PhoneStatus is used when `VoIPPhone.stop()` has been called, but has not finished stopping.

PhoneStatus.FAILED
This PhoneStatus is used when `VoIPPhone.start()` has been called, but failed to start due to an error.

3.3 Classes

3.3.1 VoIPCall

The VoIPCall class is used to represent a single VoIP session, which may be to multiple *clients*.

class **pyVoIP.VoIP.call.VoIPCall(phone: *VoIPPhone*, callstate: *CallState*, request: *SIPMessage*, session_id: int, bind_ip: str, conn: *VoIPConnection*, ms: Optional[Dict[int, *PayloadType*]] = None, sendmode=”sendonly”)**

The *phone* argument is the initating instance of *VoIPPhone*.

The *callstate* arguement is the initiating *CallState*.

The *request* argument is the *SIPMessage* representation of the SIP INVITE request from the VoIP server.

The *session_id* argument is a unique code used to identify the session with SDP when answering the call.

The *bind_ip* argument is the IP address that pyVoIP will bind its sockets to.

The *ms* arguement is a dictionary with int as the key and a *PayloadType* as the value. This is only used when originating the call.

get_dtmf(length=1) -> str
This method can be called get the next pressed DTMF key. DTMF’s are stored in an StringIO which is a buffer. Calling this method when there a key has not been pressed returns an empty string. To return the entire contents of the buffer set length to a negative number or None. If the *client* presses the numbers 1-9-5 you’ll have the following output:

```
self.get_dtmf()
>>> '1'
self.get_dtmf(length=2)
>>> '95'
self.get_dtmf()
>>> ''
```

answer() -> None
Answers the call if the phone's state is CallState.RINGING.

transfer(user: Optional[str] = None, uri: Optional[str] = None, blind=True) -> bool
Sends a REFER request to transfer the call. If blind is true (default), the call will immediately end after received a 200 or 202 response. Otherwise, it will wait for the Transferee to report a successful transfer. Or, if the transfer is unsuccessful, the call will continue. This function returns true if the transfer is blind or successful, and returns false if it is unsuccessful.

If using a URI to transfer, you must use a complete URI to include <> brackets as necessary.

ringing(request: *SIPMessage*) -> None
This function is what is called when receiving a new call. Custom VoIPCall classes should override this function to answer the call.

deny() -> None
Denies the call if the phone's state is CallState.RINGING.

hangup() -> None
Ends the call if the phone's state is CallState.ANSWRED.

cancel() -> None
Cancels a dialing call.

write_audio(data: bytes) -> None
Writes linear/raw audio data to the transmit buffer before being encoded and sent. The *data* argument MUST be bytes. **This audio must be linear/not encoded.** *RTPClient* **will encode it before transmitting.**

read_audio(length=160, blocking=True) -> bytes
Reads linear/raw audio data from the received buffer. Returns *length* amount of bytes. Default length is 160 as that is the amount of bytes sent per PCMU/PCMA packet. When *blocking* is set to true, this function will not return until data is available. When *blocking* is set to false and data is not available, this function will return `b"\x80" * length`.

3.3.2 VoIPPhoneParameter

class **pyVoIP.VoIP.phone.VoIPPhoneParameter(server: str, port: int, user: str, credentials_manager: Optional[*CredentialsManager*], bind_ip="0.0.0.0", bind_port=5060, bind_network="0.0.0.0/0", hostname: Optional[str] = None, remote_hostname: Optional[str] = None, transport_mode=*TransportMode*.UDP, cert_file: Optional[str] = None, key_file: Optional[str] = None, key_password: *KEY_PASSWORD* = None, rtp_port_low=10000, rtp_port_high=20000, call_class: Type[VoIPCall] = None, sip_class: Type[SIP.SIPClient] = None)**
The *server* argument is your PBX/VoIP server's IP.

The *port* argument is your PBX/VoIP server's port.

The *user* argument is the user element of the URI. This MAY not be the username which is used for authentication.

The *credentials_manager* argument is a *CredentialsManager* instance that stores all usernames and passwords your phone may need.

The *bind_ip* argument is used to bind SIP and RTP ports to receive incoming calls. Default is to bind to 0.0.0.0, however, this is not recommended.

The *bind_port* argument is the port SIP will bind to to receive SIP requests. The default for this protocol is port 5060, but any port can be used.

The *bind_network* argument is used to configure pyVoIP's NAT. pyVoIP uses this to know whether to use the *hostname* or *remote_hostname* when generating SIP requests to in-network and out-of-network devices respectively. Value must be a string with IPv4 CIDR notation.

The *hostname* argument is used to generate SIP requests and responses with devices within pyVoIP's *bind_network*. If left as None, the *bind_ip* will be used instead.

The *remote_hostname* argument is used to generate SIP requests and responses with devices outside of pyVoIP's *bind_network*. If left as None, pyVoIP will throw a *NATError* if a request is sent outside of pyVoIP's *bind_network*.

The *transport_mode* argument determines whether pyVoIP will use UDP, TCP, or TLS. Value should be a *TransportMode*.

The *cert_file*, *key_file*, and *key_password* arguments are used to load certificates in pyVoIP's server context if using TLS for the transport mode. See Python's documentation on load_cert_chain for more details.

The *rtp_port_low* and *rtp_port_high* arguments are used to generate random ports to use for audio transfer. Per RFC 4566 Sections 5.7 and 5.14, it can take multiple ports to fully communicate with other *clients*, as such a large range is recommended. If an invalid range is given, a *InvalidStateError* will be thrown.

The *call_class* argument allows to override the used *VoIPCall* class (must be a child class of *VoIPCall*).

The *sip_class* argument allows to override the used *SIPClient* class (must be a child class of *SIPClient*).

3.3.3 VoIPPhone

The VoIPPhone class is used to manage the *SIPClient* class and create *VoIPCall*'s when there is an incoming call or a *user* makes a call. It then uses the VoIPCall class to handle the call's states.

***class* pyVoIP.VoIP.phone.VoIPPhone(voip_phone_parameter: *VoIPPhoneParameter*)**

get_status() -> *PhoneStatus*
This method returns the phone's current status.

start() -> None
This method starts the *SIPClient* class. On failure, this will automatically call stop().

stop() -> None
This method ends all ongoing calls, then stops the *SIPClient* class

call(number: str, payload_types: Optional[List[*PayloadType*]] = None) -> *VoIPCall*
Originates a call using the specified *payload_types*, or PCMU and telephone-event by default. The *number* argument must be a string.

Returns a *VoIPCall* class in CallState.DIALING.

message(number: str, body: str, ctype = "text/plain") -> bool
Sends a MESSAGE request to the *number* with the text of *body*, and the Content-Type header is set to the value of *ctype*.

CHAPTER

FOUR

CREDENTIALS

Since SIP requests can traverse multiple servers and can receive multiple challenges, the Credentials Manager was made to store multiple passwords and pyVoIP will use the appropriate password upon request.

Per RFC 3261 Section 22.1, SIP uses authentication similar to HTTP authentication (**RFC 2617**), with the main difference being `The realm string alone defines the protection domain.`. However, some services always use the same domain. For example, if you need to authenticate with two seperate Asterisk servers, the realm will almost certainly be `asterisk` for both, despite being otherwise unrelated servers. For that reason, the Credentials Manager also supports server filtering.

4.1 CredentialsManager

***class* pyVoIP.credentials.CredentialsManager()**

add(username: str, password: str, server: Optional[str] = None, realm: Optional[str] = None, user: Optional[str] = None) -> None

This method registers a username and password combination with the Credentials Manager.

The *username* argument is the username that will be used in the Authentication header and digest calculation.

The *password* argument is the password that will be used in the Authentication header digest calculation.

The *server* argument is used to determine the correct credentials when challenged for authentication. If *server* is left as `None`, the credentials may be selected with any server.

The *realm* argument is used to determine the correct credentials when challenged for authentication. If *realm* is left as `None`, the credentials may be selected with any realm.

The *user* argument is used to determine the correct credentials when challenged for authentication. If *user* is left as `None`, the credentials may be selected with any user. The *user* argument is the user in the SIP URI, **not** the username used in authentication.

get(server: str, realm: str, user: str) -> Dict[str, str]

Looks for credentials that match the server, realm, and user in that order. If no matchng credentials are found, this will return anonymous credentials as a server MAY accept them per RFC 3261 Section 22.1.

CHAPTER

FIVE

SIP - SESSION INITIATION PROTOCOL

The SIP module receives, parses, and responds to incoming SIP requests and responses. If appropriate, it then forwards them to the *VoIPPhone*.

5.1 Errors

There are two errors under `pyVoIP.SIP.error`.

exception **pyVoIP.SIP.InvalidAccountInfoError**

This is thrown when *SIPClient* gets a bad response when trying to register with the PBX/VoIP server. This error also kills the SIP REGISTER thread, so you will need to call SIPClient.stop() then SIPClient.start().

exception **pyVoIP.SIP.SIPParseError**

This is thrown when *SIPMessage* is unable to parse a SIP message/request.

5.2 Enums

pyVoIP.SIP.message.SIPMessageType

SIPMessageType is an IntEnum with two members. It's stored in `SIPMessage.type` to effectively parse the message.

SIPMessageType.REQUEST

This SIPMessageType is used to signify the message was a SIP request.

SIPMessageType.RESPONSE

This SIPMessageType is used to signify the message was a SIP response.

pyVoIP.SIP.message.SIPStatus

SIPStatus is used for *SIPMessage*'s with SIPMessageType.RESPONSE. They will not all be listed here, but a complete list can be found on Wikipedia. SIPStatus has the following attributes:

status.value

This is the integer value of the status. For example, `SIPStatus.OK.value` is equal to `int(200)`.

status.phrase

This is the string value of the status, usually written next to the number in a SIP response. For example, `SIPStatus.TRYING.phrase` is equal to `"Trying"`.

status.description

This is the string value of the description of the status, it can be useful for debugging. For example, `SIPStatus.OK.description` is equal to `"Request successful"` Not all responses have a description.

Here are a few common SIPStatus members and their attributes in the order of value, phrase, description:

SIPStatus.TRYING
100, "Trying", "Extended search being performed, may take a significant time"

SIPStatus.RINGING
180, "Ringing", "Destination user agent received INVITE, and is alerting user of call"

SIPStatus.OK
200, "OK", "Request successful"

SIPStatus.BUSY_HERE
486, "Busy Here", "Callee is busy"

5.3 Classes

5.3.1 SIPClient

The SIPClient class is used to communicate with the PBX/VoIP server. It is responsible for registering with the server, and receiving phone calls.

***class* pyVoIP.SIP.client.SIPClient(server: str, port: int, user: str, credentials_manager: *CredentialsManager*, bind_ip="0.0.0.0", bind_network="0.0.0.0/0", hostname: Optional[str] = None, remote_hostname: Optional[str] = None, bind_port=5060, call_callback: Optional[Callable[[*VoIPConnection*, *SIPMessage*], Optional[str]]] = None, fatal_callback: Optional[Callable[..., None]] = None, transport_mode: *TransportMode* = TransportMode.UDP, cert_file: Optional[str] = None, key_file: Optional[str] = None, key_password: *KEY_PASSWORD* = None)**

The *server* argument is your PBX/VoIP server's IP.

The *port* argument is your PBX/VoIP server's port.

The *user* argument is the user element of the URI. This MAY not be the username which is used for authentication.

The *credentials_manager* argument is a *CredentialsManager* instance that stores all usernames and passwords your phone may need.

The *bind_ip* argument is used to bind SIP and RTP ports to receive incoming calls. Default is to bind to 0.0.0.0, however, this is not recommended.

The *bind_network* argument is used to configure pyVoIP's NAT. pyVoIP uses this to know whether to use the *hostname* or *remote_hostname* when generating SIP requests to in-network and out-of-network devices respectively. Value must be a string with IPv4 CIDR notation.

The *hostname* argument is used to generate SIP requests and responses with devices within pyVoIP's *bind_network*. If left as None, the *bind_ip* will be used instead.

The *remote_hostname* argument is used to generate SIP requests and responses with devices outside of pyVoIP's *bind_network*. If left as None, pyVoIP will throw a *NATError* if a request is sent outside of pyVoIP's *bind_network*.

The *bind_port* argument is the port SIP will bind to to receive SIP requests. The default for this protocol is port 5060, but any port can be used.

The *call_callback* argument is a function that tells the *VoIPPhone* instance it is receiving a call.

The *fatal_callback* argument is a function that tells the *VoIPPhone* instance there was a fatal error, e.g., failed to register.

The *transport_mode* argument determines whether pyVoIP will use UDP, TCP, or TLS. Value should be a *TransportMode*.

The *cert_file*, *key_file*, and *key_password* arguments are used to load certificates in pyVoIP's server context if using TLS for the transport mode. See Python's documentation on load_cert_chain for more details.

start() -> None

This method starts the SIPClient and registers with the PBX/VoIP server. It is called automatically when *VoIPPhone* starts.

stop() -> None

This method stops the SIPClient and deregisters with the PBX/VoIP server. It is called automatically when *VoIPPhone* stops.

send(request: str) -> ***VoIPConnection***

This method starts a new SIP dialog and sends the request using the request to determine its destination. Returns the VoIPConnection to continue the dialog.

invite(number: str, ms: dict[int, dict[str, ***PayloadType*****]], sendtype:** ***TransmitType*****)**

This method generates a SIP INVITE request. This method is called by *VoIPPhone*.call().

The *number* argument must be the number being called as a string.

The *ms* argument is a dictionary of the media types to be used. Currently only PCMU and telephone-event is supported.

The *sendtype* argument must be an instance of *TransmitType*.

bye(request: ***SIPMessage*****) -> None**

This method is called by *VoIPCall*.hangup(). It generates a BYE request, and then transmits the generated request. **This should not be called by the** *user*.

deregister() -> bool

This method is called by SIPClient.stop() after the REGISTER thread is stopped. It will generate and transmit a REGISTER request with an Expiration of zero. Telling the PBX/VoIP server it is turning off. **This should not be called by the** *user*.

register() -> bool

This method is called by the REGISTER thread. It will generate and transmit a REGISTER request telling the PBX/VoIP server that it will be online for at least 300 seconds. The REGISTER thread will call this function every 295 seconds. **This should not be called by the** *user*.

5.3.2 SIPMessage

The SIPMessage class is used to parse SIP requests and responses and makes them easily processed by other classes.

class **pyVoIP.SIP.message.SIPMessage(data: bytes)**

The *data* argument is the SIP message in bytes. It is then passed to SIPMessage.parse().

SIPMessage has the following attributes:

SIPMessage.heading

This attribute is the first line of the SIP message as a string. It contains the SIP Version, and the method/response code.

SIPMessage.type

This attribute will be a *SIPMessageType*.

SIPMessage.status

This attribute will be a *SIPStatus*. It will be set to `int(0)` if the message is a request.

SIPMessage.method

This attribute will be a string representation of the method. It will be set to None if the message is a response.

SIPMessage.headers

This attribute is a dictionary of all the headers in the request, and their parsed values.

SIPMessage.body

This attribute is a dictionary of the content of the body.

SIPMessage.authentication

This attribute is a dictionary of a parsed Authentication header. There are two authentication headers: Authorization, and WWW-Authenticate. See RFC 3261 Sections 20.7 and 20.44 respectively.

SIPMessage.raw

This attribute is an unparsed version of the *data* argument, in bytes.

summary() -> str

This method returns a string representation of the SIP request.

CHAPTER

SIX

RTP - REAL-TIME TRANSPORT PROTOCOL

The RTP module recives and transmits sound and phone-event data for a particular phone call.

6.1 Functions

The RTP module has two functions that are used by various classes for packet parsing.

pyVoIP.RTP.byte_to_bits(byte: bytes) -> str
: This method converts a single byte into an eight character string of ones and zeros. The *byte* argument must be a single byte.

pyVoIP.RTP.add_bytes(bytes: bytes) -> int
: This method takes multiple bytes and adds them together into an integer.

6.2 Errors

exception **pyVoIP.RTP.DynamicPayloadType**
: This may be thrown when you try to int cast a dynamic PayloadType. Most PayloadTypes have a number assigned in RFC 3551 Section 6. However, some are considered to be ‘dynamic’ meaning the PBX/VoIP server will pick an available number, and define it.

exception **pyVoIP.RTP.RTPParseError**
: This is thrown by *RTPMessage* when unable to parse a RTP message. It may also be thrown by *RTPClient* when it’s unable to encode or decode the RTP packet payload.

6.3 Enums

pyVoIP.RTP.RTPProtocol
: RTPProtocol is an Enum with three attributes. It defines the method that packets are to be sent with. Currently, only AVP is supported.

  RTPProtocol.UDP
  : This means the audio should be sent with pure UDP. Returns `'udp'` when string casted.

  RTPProtocol.AVP
  : This means the audio should be sent with RTP Audio/Video Protocol described in **RFC 3551**. Returns `'RTP/AVP'` when string casted.

RTPProtocol.SAVP
This means the audio should be sent with RTP Secure Audio/Video Protocol described in **RFC 3711**. Returns `'RTP/SAVP'` when string casted.

pyVoIP.RTP.TransmitType
TransmitType is an Enum with four attributes. It describes how the *RTPClient* should act.

TransmitType.RECVONLY
This means the RTPClient should only recive audio, not transmit it. Returns `'recvonly'` when string casted.

TransmitType.SENDRECV
This means the RTPClient should send and receive audio. Returns `'sendrecv'` when string casted.

TransmitType.SENDONLY
This means the RTPClient should only send audio, not receive it. Returns `'sendonly'` when string casted.

TransmitType.INACTIVE
This means the RTP client should not send or receive audio, and instead wait to be activated. Returns `'inactive'` when string casted.

pyVoIP.RTP.PayloadType
PayloadType is an Enum with multiple attributes. It described the list of attributes in RFC 3551 Section 6. Currently, only one dynamic event is assigned: telephone-event. Telephone-event is used for sending and recieving DTMF codes. There are a few conflicing names in the RFC as they're the same codec with varrying options so we will go over the conflicts here. PayloadType has the following attributes:

type.value
This is either the number assigned as PT in the RFC 3551 Section 6 chart, or it is the encoding name if it is dynamic. Int casting the PayloadType will return this number, or raise a DynamicPayloadType error if the protocol is dynamic.

type.rate
This will return the clock rate of the codec.

type.channel
This will return the number of channels the used in the codec, or for Non-codecs like telephone-event, it will return zero.

type.description
This will return the encoding name of the payload. String casting the PayloadType will return this value.

PayloadType.DVI4_8000
This variation of the DVI4 Codec has the attributes: value 5, rate 8000, channel 1, description "DVI4"

PayloadType.DVI4_16000
This variation of the DVI4 Codec has the attributes: value 6, rate 16000, channel 1, description "DVI4"

PayloadType.DVI4_11025
This variation of the DVI4 Codec has the attributes: value 16, rate 11025, channel 1, description "DVI4"

PayloadType.DVI4_22050
This variation of the DVI4 Codec has the attributes: value 17, rate 22050, channel 1, description "DVI4"

PayloadType.L16
This variation of the L16 Codec has the attributes: value 11, rate 44100, channel 1, description "L16"

PayloadType.L16_2
This variation of the L16 Codec has the attributes: value 11, rate 44100, channel 2, description "L16"

PayloadType.EVENT
This is the dynamic non-codec 'telephone-event'. Telephone-event is used for sending and receiving DTMF codes.

6.4 Classes

6.4.1 RTPPacketManager

The RTPPacketManager class utilizes an `io.ByteIO` that stores either received payloads, or raw audio data waiting to be transmitted.

pyVoIP.RTP.**RTPPacketManager**()

read(length=160) -> bytes
Reads *length* bytes from the ByteIO. This will always return the length requested, and will append `b'\x80'`'s onto the end of the available bytes to achieve this length.

rebuild(reset: bool, offset=0, data=b'') -> None
This rebuilds the ByteIO if packets are sent out of order. Setting the argument *reset* to `True` will wipe all data in the ByteIO and insert in the data in the argument *data* at the position in the argument *offset*.

write(offset: int, data: bytes) -> None
Writes the data in the argument *data* to the ByteIO at the position in the argument *offset*. RTP data comes with a timestamp that is passed as the offset in this case. This makes it so a hole left by delayed packets can be filled later. If a packet with a timestamp sooner than any other timestamp received, it will rebuild the ByteIO with the new data. If this new position is over 100,000 bytes before the earliest byte, the ByteIO is completely wiped and starts over. This is to prevent Overflow errors.

6.4.2 RTPMessage

The RTPMessage class is used to parse RTP packets and makes them easily processed by the *RTPClient*.

pyVoIP.RTP.**RTPMessage**(data: bytes, assoc: dict[int, *PayloadType*])

The *data* argument is the received RTP packet in bytes.

The *assoc* argument is a dictionary, using the payload number as a key and a *PayloadType* as the value. This way RTPMessage can determine what number a dynamic payload is. This association dictionary is generated by *VoIPCall*.

RTPMessage has attributes that come from RFC 3550 Section 5.1. RTPMessage has the following attributes:

RTPMessage.version
This attribute is the RTP packet version, represented as an integer.

RTPMessage.padding
If this attribute is set to True the payload has padding.

RTPMessage.extension
If this attribute is set to True the packet has a header extension.

RTPMessage.CC
This attribute is the CSRC Count, represented as an integer.

RTPMessage.marker
This attribute is set to True if the marker bit is set.

RTPMessage.payload_type
This attribute is set to the *PayloadType* that corresponds to the payload codec.

RTPMessage.sequence
This attribute is set to the sequence number of the RTP packet, represented as an integer.

RTPMessage.timestamp
This attribute is set to the timestamp of the RTP packet, represented as an integer.

RTPMessage.SSRC
This attribute is set to the synchronization source of the RTP packet, represented as an integer.

RTPMessage.payload
This attribute is the payload data of the RTP packet, represented as bytes.

RTPMessage.raw
This attribute is the unparsed version of the *data* argument, in bytes.

summary() -> str
This method returns a string representation of the RTP packet excluding the payload.

parse(data: bytes) -> None
This method is called by the initialization of the class. It determines the RTP version, whether the packet has padding, has a header extension, and other information about the backet.

6.4.3 RTPClient

The RTPClient is used to send and receive RTP packets and encode/decode the audio codecs.

class pyVoIP.RTP.**RTPClient**(assoc: dict[int, *PayloadType*], inIP: str, inPort: int, outIP: str, outPort: int, sendrecv: *TransmitType*, dtmf: Optional[Callable[[str], None] = None):

The *assoc* argument is a dictionary, using the payload number as a key and a *PayloadType* as the value. This way, RTPMessage can determine what a number a dynamic payload is. This association dictionary is generated by *VoIPCall*.

The *inIP* argument is used to receive incoming RTP message.

The *inPort* argument is the port RTPClient will bind to, to receive incoming RTP packets.

The *outIP* argument is used to transmit RTP packets.

The *outPort* argument is used to transmit RTP packets.

The *sendrecv* argument describes how the RTPClient should act. Please reference *TransmitType* for more details.

The *dtmf* argument is set to the callback *VoIPCall*.dtmfCallback().

start() -> None
This method is called by *VoIPCall*.answer(). It starts the recv() and trans() threads. It is also what initiates the bound port. **This should not be called by the** *user*.

stop() -> None
This method is called by *VoIPCall*.hangup() and *VoIPCall*.bye(). It stops the recv() and trans() threads. It will also close the bound port. **This should not be called by the** *user*.

read(length=160, blocking=True) -> bytes
This method is called by *VoIPCall*.readAudio(). It reads linear/raw audio data from the received buffer. Returns *length* amount of bytes. Default length is 160 as that is the amount of bytes sent per

PCMU/PCMA packet. When *blocking* is set to true, this function will not return until data is available. When *blocking* is set to false and data is not available, this function will return bytes(length).

write(data: bytes) -> None
This method is called by *VoIPCall*.writeAudio(). It queues the data written to be sent to the *client*.

recv() -> None
This method is called by RTPClient.start() and is responsible for receiving and parsing through RTP packets. **This should not be called by the** *user*.

trans() -> None
This method is called by RTPClient.start() and is responsible for transmitting RTP packets. **This should not be called by the** *user*.

parse_packet(packet: bytes) -> None
This method is called by the recv() thread. It converts the argument *packet* into a *RTPMessage*, then sends it to the proper parse function depending on the *PayloadType*.

encode_packet(payload: bytes) -> bytes
This method is called by the trans() thread. It encoded the argument *payload* into the prefered codec. Currently, PCMU is the hardcoded prefered codec. The trans() thread will use the payload to create the RTP packet before transmitting.

parse_pcmu(packet: *RTPMessage*) -> None
This method is called by parse_packet(). It will decode the *packet*'s payload from PCMU to linear/raw audio and write it to the incoming *RTPPacketManager*.

encode_pcmu(packet: bytes) -> bytes
This method is called by encode_packet(). It will encode the *payload* into the PCMU audio codec.

parse_pcma(packet: *RTPMessage*) -> None
This method is called by parse_packet(). It will decode the *packet*'s payload from PCMA to linear/raw audio and write it to the incoming *RTPPacketManager*.

encode_pcma(payload: bytes) -> bytes
This method is called by encode_packet(). It will encode the *payload* into the PCMA audio codec.

parse_telephone_event(packet: *RTPMessage*) -> None
This method is called by parse_packet(). It will decode the *packet*'s payload from the telephone-event non-codec to the string representation of the event. It will then call *VoIPCall*.dtmf_callback().

CHAPTER

SEVEN

NETWORKING

VoIP uses a lot of complex networking in order to accomplish its tasks. The networking module handles all of these complex operations.

7.1 Errors

exception **pyVoIP.networking.nat.NATError**

This is thrown when *NAT* is either unable to resolve a hostname or when a remote hostname has not been specified and an attempt was made to connect to a remote host.

7.2 Enums

pyVoIP.networking.nat.AddressType

Used for determining remote or local tags in SIP messages.

AddressType.**REMOTE**

AddressType.**LOCAL**

pyVoIP.networking.transport.TransportMode

TransportMode is used by pyVoIP to determine what communication protocol to use. TransportMode has the following properties:

TransportMode.value

This is the string value of the TransportMode. For example, `UDP` or `TLS`.

TransportMode.socket_type

This is the SocketKind associated with the TransportMode.

TransportMode.tls_mode

This is either PROTOCOL_TLS when using TLS or None otherwise.

Here is a list of current supported transport modes:

TransportMode.UDP

“UDP”, SOCK_DGRAM, None

TransportMode.TCP

“TCP”, SOCK_STREAM, None

TransportMode.TLS

“TLS”, SOCK_STREAM, PROTOCOL_TLS

7.3 Classes

7.3.1 NAT

The NAT class is used automatically understand and translate IPs and hostnames for LAN to WAN connections and vice versa.

class **pyVoIP.networking.nat.NAT(bind_ip: str, network: str, hostname: Optional[str] = None, remote_hostname: Optional[str] = None)**

The *bind_ip* argument is the IP address that pyVoIP will bind its sockets to.

The *network* argument is used to know whether to use the *hostname* or *remote_hostname* when generating SIP requests to in-network and out-of-network devices respectively. Value must be a string with IPv4 CIDR notation.

The *hostname* argument is used to generate SIP requests and responses with devices within pyVoIP's *bind_network*. If left as None, the *bind_ip* will be used instead.

The *remote_hostname* argument is used to generate SIP requests and responses with devices outside of pyVoIP's *bind_network*. If left as None, pyVoIP will throw a *NATError* if a request is sent outside of pyVoIP's *bind_network*.

get_host(host: str) -> str

This method return the hostname another *client* needs to connect to us.

check_host(host: str) -> ***AddressType***

This method determine if a host is a remote computer or not.

7.3.2 VoIPSocket

The VoIPSocket class is the phone's main SIP socket. It receives and processes all new dialogs, and all messages if using *TransportMode*.UDP.

class **pyVoIP.networking.sock.VoIPSocket(mode:** ***TransportMode*****, bind_ip: str, bind_port: int, sip:** ***SIPClient*****, cert_file: Optional[str] = None, key_file: Optional[str] = None, key_password:** ***KEY_PASSWORD*** **= None)**

The *TransportMode* argument is used to determine what communication protocol to use.

The *bind_ip* argument is the IP address that pyVoIP will bind its sockets to.

The *bind_port* argument is the port SIP will bind to to receive SIP requests.

The *sip* argument is a *SIPClient* instance reference.

The *cert_file*, *key_file*, and *key_password* arguments are used to load certificates in pyVoIP's server context if using TLS for the transport mode. See Python's documentation on load_cert_chain for more details.

get_database_dump(pretty=False) -> str

If using UDP, all messages and dialog states are stored in an in-memory sqlite3 database. This function will return a string with all entries from the dialogs (listening) table and unread messages (msgs) table. If *pretty* is set to true, it will use Python's pprint module to make the test reader friendly for a print statement. If *pretty* is set to false, it will return JSON instead.

send(data: bytes) -> ***VoIPConnection***

Creates a new connection / dialog, sends the data, then returns the socket.

7.3.3 VoIPConnection

The VoIPConnecion class is a wrapper for Python's sockets. Since UDP, TCP, and TLS sockets all have different quarks in Python, this class consolidates everything into one interface. For UDP, VoIPConnection will pull messages from *VoIPSocket*'s database.

class **pyVoIP.networking.sock.VoIPConnection(voip_sock: *VoIPSocket*, conn: Optional[*SOCKETS*, message: *SIPMessage*)**

The *voiop_socket* argument is a *VoIPSocket* instance reference.

The *conn* argument is the underlying Python socket.

The *message* argument is the *SIPMessage* used to initate the dialog.

send(data: Union[bytes, str]) -> None

Sends *data* to the *client*. If *data* is a string, it will be UTF8 encoded first.

peak() -> bytes

Calls `recv` with *peak* set to true.

recv(nbytes=8192, timeout=0, peak=False) -> bytes

Receives the next *nbytes* from the socket. The *timeout* argument is in seconds, and if set to `0` it will not timeout. If the *peak* argument is set to True, it will receive the next *nbytes* from the socket and return them, however, the same data will be returned upon the next call of `recv`.

close() -> None

Closes the socket.

CHAPTER

EIGHT

TYPES

pyVoIP has several type aliases that it stores in `pyVoIP.types`.

pyVoIP.types.URI_HEADER = Dict[str, Union[str, int]]

This is for URI Headers (such as To, From, Contact, etc) dictionaries in a *SIPMessage*.

pyVoIP.types.SOCKETS = Union[socket.socket, ssl.SSLSocket]

This is in a few places in *VoIPSocket* and *VoIPConnection*.

pyVoIP.types.KEY_PASSWORD = Union[bytes, bytearray, str, Callable[[], bytes], Callable[[], bytearray], Callable[[], str]]

This is used for TLS settings. See Python's documentation on load_cert_chain's password argument for more details.

pyVoIP.types.CREDENTIALS_DICT = Dict[Optional[str], Dict[Optional[str], Dict[Optional[str], Dict[str, str]]]]

This is the format of the *CredentialsManager*'s internal dictionary.

INDEX