pyVoIP

Release 1.6.8

Tayler J Porter

Jan 17, 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 callback functions to initiate phone calls. In the example below, our callback function is named `answer`. The callback takes one argument, which is a *VoIPCall* instance.

We are also importing *VoIPPhone* and *InvalidStateError*. VoIPPhone is the main class for our softphone. 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.VoIP import VoIPPhone, InvalidStateError

def answer(call):
    try:
        call.answer()
        call.hangup()
    except InvalidStateError:
        pass

if __name__ == "__main__":
    phone = VoIPPhone(<SIP server IP>, <SIP server port>, <SIP server username>, <SIP␣
↪server password>, myIP=<Your computer's local IP>, callCallback=answer)
    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*.

```
from pyVoIP.VoIP import VoIPPhone, InvalidStateError, CallState
import time
import wave
```

(continues on next page)

(continued from previous page)

```
def answer(call):
    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__":
    phone = VoIPPhone(<SIP Server IP>, <SIP Server Port>, <SIP Server Username>, <SIP␣
↪Server Password>, myIP=<Your computers local IP>, callCallback=answer)
    phone.start()
    input('Press enter to disable the phone')
    phone.stop()
```

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

```
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.

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:

```
from pyVoIP.VoIP import VoIPPhone, InvalidStateError, CallState
import time
import wave

def answer(call):
    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":
                # Do something
                call.hangup()
            elif dtmf == "2":
                # Do something else
                call.hangup()
            time.sleep(0.1)
    except InvalidStateError:
        pass
    except:
        call.hangup()

if __name__ == '__main__':
    phone = VoIPPhone(<SIP Server IP>, <SIP Server Port>, <SIP Server Username>, <SIP␣
↪Server Password>, myIP=<Your computers local IP>, callCallback=answer)
    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.

CHAPTER

TWO

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 open a GitHub issue. Otherwise, does nothing.

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.

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 use the SIP and RTP modules independently and create your own version of the VoIP module. If you choose to use the VoIP module, this section will explain how.

3.1 Errors

There are two errors under `pyVoIP.VoIP`.

***exception* VoIP.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 been ended.

***exception* VoIP.InvalidRangeError**
This is thrown by *VoIPPhone* when you define the rtpPort ranges as rtpPortLow > rtpPortHigh. 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* VoIP.NoPortsAvailableError**
This is thrown when a call is attempting to be initiated but no ports are available.

3.2 Enums

VoIP.CallState
CallState is an Enum with four 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.

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

In this state, you can not use any functions.

VoIP.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.

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* VoIP.VoIPCall(phone: *VoIPPhone*, request: *SIPMessage*, session_id: int, myIP: str, rtpPortLow: int, rtpPortHigh: int)**

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 *myIP* argument is the IP address it will pass to *RTPClient*'s to bind 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.

dtmfCallback(code: str) -> None
Deprecated. Please use `dtmf_callback` instead.

dtmf_callback(code: str) -> None
This method is called by *RTPClient*'s when a telephone-event DTMF message is received. The *code* argument is a string. It should be an Event in complinace with RFC 4733 Section 3.2.

getDTMF(length=1) -> str
Deprecated. Please use `get_dtmf` instead.

get_dtmf(length=1) -> str
This method can be called get the next pressed DTMF key. DTMF's are stored in an `io.StringIO` and act as a stack. Meaning if the *client* presses the numbers 1-9-5 you'll have the following output:

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

As you can see, calling this method when there a key has not been pressed returns an empty string.

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

answered(request: *SIPMessage*) -> None
This function is called by *SIPClient* when a call originated by the *user* has been answered by the *client*.

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.

bye() -> None
Ends the call but does not send a SIP BYE message to the SIP server. This function is used to end the call on the server side when the client ended the call. **THE *USER* SHOUND NOT CALL THIS FUNCTION OR THE *CLIENT* WILL BE LEFT ON THE LINE WITH NO RESPONSE. CALL HANGUP() INSTEAD.**

writeAudio(data: bytes) -> None
Deprecated. Please use `write_audio` instead.

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.**

readAudio(length=160, blocking=True) -> bytes
Deprecated. Please use `read_audio` instead.

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 VoIPPhone

The VoIPPhone class is used to manage the *SIPClient* class and create *VoIPCall*'s when there is an incoming call. It then passes the VoIPCall as the argument in the callback.

***class* VoIP.VoIPPhone(server: str, port: int, username: str, password: str, callCallback: Optional[Callable] = None, myIP: Optional[str] = None, sipPort=5060, rtpPortLow=10000, rtpPortHigh=20000)**

The *server* argument is your PBX/VoIP server's IP, represented as a string.

The *port* argument is your PBX/VoIP server's port, represented as an integer.

The *username* argument is your SIP account username on the PBX/VoIP server, represented as a string.

The *password* argument is your SIP account password on the PBX/VoIP server, represented as a string.

The *callCallback* argument is your callback function that VoIPPhone will run when you receive a call. The callback must take one argument, which will be a *VoIPCall*. If left as None, the VoIPPhone will automatically respond to all incoming calls as Busy.

The *myIP* argument is used to bind SIP and RTP ports to receive incoming calls. If left as None, the VoIPPhone will bind to 0.0.0.0.

The *sipPort* 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 *rtpPortLow* and *rtpPortHigh* 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.

callback(request: *SIPMessage*) -> None
This method is called by the *SIPClient* when an INVITE or BYE request is received. This function then creates a *VoIPCall* or terminates it respectively. When a VoIPCall is created, it will then pass it to the *callCallback* function as an argument. If *callCallback* is set to None, this function replies as BUSY. **This function should not be called by the** *user*.

getStatus() -> PhoneStatus
Deprecated. Please use `get_status` instead.

get_status() -> PhoneStatus
This method returns the PhoneStatus.

request_port(blocking=True) -> int
This method is called when a new port is needed to use in a *VoIPCall*. If blocking is set to True, this will wait until a port is available. Otherwise, it will raise NoPortsAvailableError.

release_ports(call: Optional[VoIPCall] = None) -> None
This method is called when a call ends. If call is provided, it will only release the ports used by that *VoIPCall*. Otherwise, it will iterate through all active calls, and release all ports that are no longer in use.

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

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

call(number: str) -> *VoIPCall*
Originates a call using PCMU and telephone-event. The *number* argument must be a string, and it returns a *VoIPCall* class in CallState.DIALING. You should use a while loop to wait until the CallState is ANSWRED.

CHAPTER

FOUR

SIP - SESSION INITIATION PROTOCOL

The SIP module receives, parses, and responds to incoming SIP requests/messages. If appropriate, it then forwards them to the *callback* method of *VoIPPhone*.

4.1 Errors

There are two errors under `pyVoIP.SIP`.

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

4.2 Enums

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

SIPMessageType.MESSAGE
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.

SIP.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' 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'

4.3 Classes

4.3.1 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* SIP.SIPClient(server: str, port: int, username: str, password: str, myIP="0.0.0.0", myPort=5060, callCallback: Optional[Callable[[SIPMessage], None]] = None)**

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

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

The *username* argument is your SIP account username on the PBX/VoIP server.

The *password* argument is your SIP account password on the PBX/VoIP server.

The *myIP* argument is used to bind a socket and receive incoming SIP requests and responses.

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

The *callCallback* argument is the callback function for *VoIPPhone*. VoIPPhone will process the SIP request, and perform the appropriate actions.

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

parseMessage(message: *SIPMessage*) -> None
Deprecated. Please use `parse_message` instead.

parse_message(message: *SIPMessage*) -> None
This method is called by SIPClient.recv() and is responsible for parsing through SIP responses. **This should not be called by the** *user*.

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

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

genCallID() -> str
Deprecated. **This should not be called by the** *user*.

gen_call_id() -> str
This method is called by other 'gen' methods when a new Call-ID header is needed. See RFC 3261 Section 20.8. **This should not be called by the** *user*.

lastCallID() -> str
Deprecated. **This should not be called by the** *user*.

last_call_id() -> str
This method is called by other 'gen' methods when the last Call-ID header is needed. See RFC 3261 Section 20.8. **This should not be called by the** *user*.

genTag() -> str
Deprecated. **This should not be called by the** *user*.

gen_tag() -> str
This method is called by other 'gen' methods when a new tag is needed. See RFC 3261 Section 8.2.6.2. **This should not be called by the** *user*.

genSIPVersionNotSupported() -> str
Deprecated. **This should not be called by the** *user*.

gen_sip_version_not_supported() -> str
This method is called by the recv() thread when it has received a SIP message that is not SIP version 2.0.

genAuthorization(request: *SIPMessage*) -> bytes
Deprecated. **This should not be called by the** *user*.

gen_authorization(request: *SIPMessage*) -> bytes
This calculates the authroization hash in response to the WWW-Authenticate header. See RFC 3261 Section 20.7. The *request* argument should be a 401 Unauthorized response. **This should not be called by the** *user*.

genRegister(request: *SIPMessage*, deregister: bool = False) -> str
Deprecated. **This should not be called by the** *user*.

gen_register(request: *SIPMessage*, deregister: bool = False) -> str
This method generates a SIP REGISTER request. The *request* argument should be a 401 Unauthorized response. If *deregister* is set to true, a SIP DE-REGISTER request is generated instead. **This should not be called by the** *user*.

genBusy(request: *SIPMessage*) -> str
Deprecated. **This should not be called by the** *user*.

gen_busy(request: *SIPMessage*) -> str
This method generates a SIP 486 'Busy Here' response. The *request* argument should be a SIP INVITE request.

genOk(request: *SIPMessage*) -> str
Deprecated. **This should not be called by the** *user*.

gen_ok(request: *SIPMessage*) -> str
This method generates a SIP 200 'Ok' response. The *request* argument should be a SIP BYE request.

genInvite(number: str, sess_id: str, ms: dict[int, dict[str, RTP.*PayloadType*]], sendtype: RTP.*RTP.TransmitType*, branch: str, call_id: str) -> str
Deprecated. **This should not be called by the** *user*.

gen_invite(number: str, sess_id: str, ms: dict[int, dict[str, RTP.*PayloadType*]], sendtype: RTP.*RTP.TransmitType*, branch: str, call_id: str) -> str
This method generates a SIP INVITE request. This is called by SIPClient.invite().

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

The *sess_id* argument must be a unique number.

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 *RTP.TransmitType*.

The *branch* argument must be a unique string starting with "z9hG4bK". See RFC 3261 Section 8.1.1.7.

The *call_id* argument must be a unique string. See RFC 3261 Section 8.1.1.4.

genRinging(request: *SIPMessage*) -> str
Deprecated. **This should not be called by the** *user*.

gen_ringing(request: *SIPMessage*) -> str
This method generates a SIP 180 'Ringing' response. The *request* argument should be a SIP INVITE request.

genAnswer(request: *SIPMessage*, sess_id: str, ms: list[dict[str, Any]], sendtype: RTP.*RTP.TransmitType*)
Deprecated. **This should not be called by the** *user*.

gen_answer(request: *SIPMessage*, sess_id: str, ms: list[dict[str, Any]], sendtype: RTP.*RTP.TransmitType*)
This method generates a SIP 200 'OK' response. Which, when in reply to an INVITE request, tells the server the *user* has answered. **This should not be called by the** *user*.

The *request* argument should be a SIP INVITE request.

The *sess_id* argument should be a string casted integer. This will be used for the SDP o tag. See RFC 4566 Section 5.2. The *sess_id* argument will also server as the <*sess-version*> argument in the SDP o tag.

The *ms* argument should be a list of parsed SDP m tags, found in the *SIPMessage*.body attribute. This is used to generate the response SDP m tags. See RFC 4566 Section 5.14.

The *sendtype* argument should be a RTP.*TransmitType* enum. This will be used to generate the SDP a tag. See RFC 4567 Section 6.

genBye(request: *SIPMessage*) -> str
Deprecated. **This should not be called by the** *user*.

gen_bye(request: *SIPMessage*) -> str
This method generates a SIP BYE request. This is used to end a call. The *request* argument should be a SIP INVITE request. **This should not be called by the** *user*.

genAck(request: *SIPMessage*) -> str
Deprecated. **This should not be called by the** *user*.

gen_ack(request: *SIPMessage*) -> str
This method generates a SIP ACK response. The *request* argument should be a SIP 401 response.

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

bye(request: *SIPMessage*) -> None
This method is called by *VoIPCall*.hangup(). It calls genBye(), 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*.

4.3.2 SIPMessage

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

***class* SIP.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 all the SDP tags in the request, and their parsed values.

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.

parse(data: bytes) -> None
This method is called by the initialization of the class. It decides the SIPMessageType, and sends it to the corresponding parse function. *Data* is the original *data* argument in the initialization of the class. **This should not be called by the** *user*.

parseSIPResponse(data: bytes) -> None
Deprecated. **This should not be called by the** *user*.

parse_sip_response(data: bytes) -> None
This method is called by parse(). It sets the *header*, *version*, and *status* attributes and may raise a *SIPParseError* if the SIP response is an unsupported SIP version. It then calls parseHeader() for each header

in the request. *Data* is the original *data* argument in the initialization of the class. **This should not be called by the** *user*.

parseSIPMessage(data: bytes) -> None
Deprecated. **This should not be called by the** *user*.

parse_sip_message(data: bytes) -> None
This method is called by parse(). It sets the *header*, *version*, and *method* attributes and may raise a *SIPParseError* if the SIP request is an unsupported SIP version. It then calls parseHeader() and parseBody() for each header or tag in the request respectively. *Data* is the original *data* argument in the initialization of the class. **This should not be called by the** *user*.

parseHeader(header: str, data: str) -> None
Deprecated. **This should not be called by the** *user*.

parse_header(header: str, data: str) -> None
This method is called by parseSIPResponse() and parseSIPMessage(). The *header* argument is the name of the header, i.e. 'Call-ID' or 'CSeq', represented as a string. The *data* argument is the value of the header, i.e. 'Ogq-T7iBmNozoUu3GL9Lvg..' or '1 INVITE', represented as a string. **This should not be called by the** *user*.

parseBody(header: str, data: str) -> None
Deprecated. **This should not be called by the** *user*.

parse_body(header: str, data: str) -> None
This method is called by parseSIPResponse() and parseSIPMessage(). The *header* argument is the name of the SDP tag, i.e. 'm' or 'a', represented as a string. The *data* argument is the value of the header, i.e. 'audio 56704 RTP/AVP 0' or 'sendrecv', represented as a string. **This should not be called by the** *user*.

CHAPTER

FIVE

RTP - REAL-TIME TRANSPORT PROTOCOL

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

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

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.

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

5.1 Errors

***exception* 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* 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.

5.2 Enums

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.

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.

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.

5.3 Classes

5.3.1 RTPPacketManager

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

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.

5.3.2 RTPMessage

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

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.

5.3.3 RTPClient

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

class 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*.

parsePacket(packet: bytes) -> None
Deprecated. Please use `parse_packet` instead.

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*.

encodePacket(payload: bytes) -> bytes
Deprecated. Please use `encode_packet` instead.

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.

parsePCMU(packet: *RTPMessage*) -> None
Deprecated. Please use `parse_pcmu` instead.

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*.

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

parsePCMA(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*.

encodePCMA(payload: bytes) -> bytes
Deprecated. Please use `encode_pcma` instead.

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

parseTelephoneEvent(packet: *RTPMessage*) -> None
Deprecated Please use `parse_telephone_event` instead.

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().

INDEX