Implementing SIP Telephony in Python

Implementer's Guide to Scalable and Robust Internet Telephony with Session Initiation Protocol in Client-Server and Peer-to-Peer modes in Python

Kundan Singh

Author's Remarks: This document is still "work in progress". Please revisit this site later to see the completed text. I plan to provide this as a free document accompanying my open-source software of the *39 Peers* project.

Copyright

Some text from IETF RFCs and Internet-Drafts are reproduced in this document to explain or assist in their implementation in accordance with the copyright notice in those RFCs and Internet-Draft. The copyright notice of those RFCs and Internet-Drafts is as follows:

Part 2 Essential SIP

In this part you will get a step-by-step implementation guide to various protocols such as SIP, SDP and RTP. It covers essential protocol suites described in RFC3261, RFC4566, RFC3264, RFC2617 and other related RFCs.

We will start with parsing and formatting of SIP addresses, then describe the parsing and formatting of SIP messages and its components. Then we will build a SIP stack API with other control functions for transaction, dialog, etc. Then we add digest authentication and other security mechanisms in our implementation. At the end of this part, you will understand how to implement the basic SIP protocol suite without worrying about client or server specific components such as media or proxy. A major part in SIP telephony implementation deals with parsing, formatting and various state machines for transactions and dialogs – which appear for both client as well as server implementations.

Addressing

Implementing URI as per RFC 2396 and SIP address

A number of aspects in SIP and related protocols use various forms of addresses. The URI or Uniform Resource Identifier is one such class of address and is defined in RFC 2396. Some example URIs are shown below:

```
sip:kundan@example.net
sip:kns10@192.168.10.20:5060;transport=tcp
http://www.39peers.net/book
```

Besides a URI, a SIP implementation also needs to deal with SIP addresses. A SIP address contains a user's display name as well as a URI as shown below. Naturally, a SIP address is a super-set of an URI as far as data information is concerned:

```
"Kundan Singh" <sip:kundan@39peers.net>
```

Dealing with addresses requires three functions: parsing, formatting and accessing the fields. We create a new module named `rfc2396` to implement these functions.

URI

Let's assume a `URI` class that represents a URI. We would want the objects of URI type to be able to interoperate with strings such that it can be parsed from a string or formatted into a string. We would also want to access the properties of the object such as `scheme`, `user`, `password`, `host` and `port`.

```
>>> u = URI('sip:kundan@example.net')
>>> print u.scheme
sip
>>> print u.host
example.net
>>> u.port = 5060
>>> print u
sip:kundan@example.net:5060
```

The object should expose the headers and parameters of the URI as well. Finally, we would want an equality test operation so that two URIs can be compared. Note that a URI comparison uses case-insensitive values for certain fields.

Let's start by defining the `URI` class.

```
class URI(object):
```

To construct this object from a string, we need to parse the string. It is possible to build a regular expression that captures most (but not all) forms of URI representations. In the simplest form the regular expression should be able to extract the scheme, user, password, host, port, parameters and headers from the string. Based on the allowed values for various parts, we can construct the regular expression as follows.

```
Import re
…
   _syntax = re.compile('^(?P<scheme>[a-zA-Z][a-zA-Z0-9\+\-\.]*):'  # scheme
         + '(?:(?:(?P<user>[a-zA-Z0-9\-\_\.\!\~\*\'\(\)&=\+\$,;\?\/\%]+)' # user
         + '(?::(?P<password>[^:@;\?]+))?)@)?' # password
         + '(?:(?:(?P<host>[^;\?:]*)(?::(?P<port>[\d]+))?))'  # host, port
         + '(?:;(?P<params>[^\?]*))?' # parameters
         + '(?:\?(?P<headers>.*))?$') # headers
```

Once a string is passed using this syntax, we can extract the various groups into appropriate properties such as `scheme`, `user`, `password`, `host`, `port`, `params` and `headers` so that these properties are available as object properties to the programmer.

```
   def __init__(self, value=''):
         m = URI._syntax.match(value)
         if not m: raise ValueError, 'Invalid URI(' + value + ')'
         self.scheme, self.user, self.password, self.host, self.port, params, headers = m.groups()
```

Now that the parsing is completed, we need to take care of the boundary cases. For example, if the string represents a "tel:" URI then actual telephone number will be in the `host` property whereas semantically it makes sense to put the phone number in the `user` property.

```
         if self.scheme == 'tel' and self.user is None:
            self.user, self.host = self.host, None
```

If the port number is empty or missing, then the port property should be set to `None` instead of an empty string. If the port number is valid then the port property should be a number instead of a string representing the number.

```
         self.port   = self.port and int(self.port) or None
```

Instead of storing the parameters and headers as single string variables named `param` and `header`, it is more convenient to create associative array for the `param` indexed by parameter name, and an array for `header` with list of header values. This allows us to access the parameters as `u.param['transport']` to access the "transport" parameter, and headers as `header[3]` to access the fourth header in the zero-bound array index.

Extracting headers string into an array is easy by splitting across "&" to get individual headers of the array.

```
         self.header = [nv for nv in headers.split('&')] if headers else []
```

To extract the parameters string into an associative array or `dict`, we need to first split across ";" then for each such string take the left side of "=" as parameter name and right side as parameter value. Note that to allow "=" in the parameter value, we cannot use `split` on individual strings; instead we use `partition`. Once we have split values, we can construct the `dict` using array of (name, value) tuples.

```
        splits = map(lambda n: n.partition('='), params.split(';')) if params else []
        self.param  = dict(map(lambda k: (k[0], k[2] if k[2] else None), splts)) if splits else {}
```

Once we are done we the parsing of the string into individual properties of this object,we construct the full constructor function by doing error checking – for the case when the value supplied is empty to construct an empty `URI` object.

```
def __init__(self, value=''):
    if value:
        ...  # parsing code
    else:
        self.scheme = self.user = self.password = self.host = self.port = None
        self.param = {};  self.header = []
```

To format the URI object as a string, we just create a string placing the individual properties appropriately. We implement the `__repr__` instead of `__str__` so that the implementation will be available to hash indexing as well. The tricky part is to construct the parameters string from the `param` property and the headers string from the `header` property. Luckily the language feature facilitates such conversion easily as shown below.

```
def __repr__(self):
    user,host = (self.user,self.host) if self.scheme != 'tel' else (None, self.user)
    return (self.scheme + ':' + ((user + \
      ((':'+self.password) if self.password else '') + '@') if user else '') + \
      (((host if host else '') + (((':'+str(self.port)) if self.port else '')) if host else '') + \
      ((';'+';'.join([(n+'='+v if v is not None else n) for n,v in self.param.items()])) if len(self.param)>0 else '') + \
      (('?'+'&'.join(self.header)) if len(self.header)>0 else '')) if self.scheme and host else '';
```

We follow a convention to implement the `dup` function for simple data objects, similar to `clone` function in java, which allows duplicating an object instance. Instead of deep copying individual properties, it is easier to just convert the object to string and back to the object for duplication.

```
def dup(self):
    return URI(self.__repr__())
```

Comparing two URI objects require us to implement two functions: `__hash__` and `__cmp__`. This gives complete order to the URI objects and hence these objects can be used as index in a table. To return the hash code for the object, we can convert it to lower-case string and return the hash of that string. Similarly, to compare the two URI objects, we can convert them to lower-case strings and compare them. Although, only certain fields in a URI specification are case in-sensitive, in our basic implementation we assume all fields are case insensitive. TODO: this may lead to some interoperability problem.

```
def __hash__(self):
    return hash(str(self).lower())
def __cmp__(self, other):
    return cmp(str(self).lower(), str(other).lower())
```

As a convenient method, we can provide a read-only property to extract the `(host, port)` tuple from the URI. This method allows us to keep host-port as a separate data type without having to look inside the tuple. For example, `u.hostPort` allows access to the tuple via the `hostPort` property.

```
@property
def hostPort(self):
    return (self.host, self.port)
```

Finally, the last property of interest is the `secure` property. Several URI schemes such as “sips”, “https” refer to the secure version of the URI scheme. Having this property allows the application to set or inspect the security level without having to know the various schemes that apply to secure URI. The following implementation as a limitation that it works only for “sips” and “https”, but can be easily extended to support other protocols. Unlike other read-write properties, the `secure` property is unique in that once set to `True` it can not be reset to `False`. Because of this uniqueness it is desirable to have a separate processing in this property instead of just storing the property as a flag.

```
def _ssecure(self, value):
    if value and self.scheme in ['sip', 'http']: self.scheme += 's'
def _gsecure(self):
    return True if self.scheme in ['sips', 'https'] else False
secure = property(fget=_gsecure, fset=_ssecure)
```

Now that we have implemented the URI class we can do some basic testing.

```
>>> print URI('sip:kundan@example.net')
sip:kundan@example.net
>>> print URI('sip:kundan:passwd@example.net:5060;transport=udp;lr?name=value&another=another')
sip:kundan:passwd@example.net:5060;lr;transport=udp?name=value&another=another
>>> print URI('sip:192.1.2.3:5060')
sip:192.1.2.3:5060
>>> print URI("sip:kundan@example.net") == URI("sip:Kundan@Example.NET")
True
>>> print 'empty=', URI()
empty=
>>> print URI('tel:+1-212-9397063')
tel:+1-212-9397063
>>> print URI('sip:kundan@192.1.2.3:5060').hostPort
('192.1.2.3', 5060)
```

Address

As mentioned before a SIP address contains a display-name and a URI. Please refer to RFC 3261 for details on the specification. It is used in various places in a SIP message, e.g., `To`, `From`, `Contact` headers.

We implement the SIP address using the `Address` class. What makes the implementation challenging is the presence of zero of more white spaces within the SIP address, optional quotes around the display name, and optional display-name property. Note that the parser should understand the following forms:

```
Kundan Singh <sip:kundan@example.net> or <sip:kundan@example.net>
"Kundan Singh" <sip:kundan@example.net>
sip:kundan@example.net
```

Likewise, there are three regular expressions to parse the address as shown below. Parsing routine can match against any of these to identify the string as a valid SIP address. The first one has no quotes around display name, the second one has quotes around display name and the third one has empty display name.

```
class Address(object):
   _syntax = [re.compile('^(?P<name>[a-zA-Z0-9\-\.\_|+|~|\t]*)<(?P<uri>[^>]+)>'),
         re.compile('^(?:"(?P<name>[a-zA-Z0-9\-\.\_|+|~|\t]+)")[\t]*<(?P<uri>[^>]+)>'),
         re.compile('^[\t]*(?P<name>)(?P<uri>[^;]+)')]
```

Let's define a method called `parse` to parse an address from string. This method matches the value against each of the above regular expressions, and if a match is found, then it extracts the display-name and URI after stripping the white-spaces.

```
   def parse(self, value):
         for s in Address._syntax:
            m = s.match(value)
            if m:
               self.displayName = m.groups()[0].strip()
               self.uri = URI(m.groups()[1].strip())
               return
```

The above method needs to be modified to accommodate certain conditions. For example, SIP defines a special SIP address of value "*" which can be present only in the `Contact` header. Secondly, a SIP message parsing routing will need to know the parts after the SIP address in various headers, e.g., the header parameters of `To` after parsing of the SIP address of `To` header. Thus, we return the number of characters parsed in this routine, so that the caller can continue beyond the SIP address.

```
   def parse(self, value):
      if str(value).startswith('*'):
         self.wildcard = True
         return 1;
      else:
         for s in Address._syntax:
            m = s.match(value)
            if m:
               self.displayName = m.groups()[0].strip()
```

```
            self.uri = URI(m.groups()[1].strip())
            return m.end()
```

Once we have the regular expression to parse, the constructor becomes straightforward.

```
def __init__(self, value=None):
    self.displayName = self.uri = None
    self.wildcard = self.mustQuote = False
    if value: self.parse(value)
```

Note the two special properties: `wildcard` and `mustQuote`. The `wildcard` property is used to indicate that the address represented by this object is a special "*" address, and the `mustQuote` property controls whether the string representation must have quoted URI even if the display name is absent.

Constructing the string representation is straightforward, as it puts the display-name and URI appropriately in the resulting string. The URI itself is represented to a string using it's `__repr__` method.

```
def __repr__(self):
    return (('"' + self.displayName + '"' + (' ' if self.uri else '')) if self.displayName else '') \
    + (('<' if self.mustQuote or self.displayName else '') \
    + repr(self.uri) \
    + ('>' if self.mustQuote or self.displayName else '')) if self.uri else ''
```

Similar to the URI class, the Address class also has the `dup` method to clone the object.

```
def dup(self):
    return Address(self.__repr__())
```

In a real-implementation of a client, sometimes it is necessary to extract the display part of the address. The specification says that the display name is optional. In such cases, the implementation uses the user part of the URI as the display text. Nevertheless, it is handy to provide a read-only `displayable` property that extracts the displayable user name from the address using some built-in criteria. The following property definition uses the first 25 characters of the display-name, user part or host part, whichever is present first in that order.

```
@property
def displayable(self):
    name = self.displayName or self.uri and self.uri.user or self.uri and self.uri.host or ''
    return name if len(name)<25 else (name[0:22] + '...')
```

Now that we are done with the basic implementation of the `Address` class, we can perform some simple tests.

```
>>> a1 = Address('"Kundan Singh" <sip:kundan@example.net>')
>>> a2 = Address('Kundan Singh   <sip:kundan@example.net>')
>>> a3 = Address('"Kundan Singh" <sip:kundan@example.net>   ')
```

```
>>> a4 = Address('<sip:kundan@example.net>')
>>> a5 = Address('sip:kundan@example.net')
>>> print str(a1) == str(a2) and str(a1) == str(a3) and str(a1.uri) == str(a4.uri) and str(a1.uri) == str(a5.uri)
True
>>> print a1
"Kundan Singh" <sip:kundan@example.net>
>>> print a1.displayable
Kundan Singh
```

isIPv4

Sometimes the processing depends on the type of address, whether the address is an IPv4 or IPv6 address. A function such as the following provides a ready-to-use utility for such checks. A simple technique is to invoke the `inet_aton` function on the data to know if the data is valid IPv4 or not. Instead of doing a `socket` call one could alternatively parse the data into individual numeric values and check the values.

```
import socket
def isIPv4(data):
   try:
      m = socket.inet_aton(data)  # alternatively: len(filter(lambda y: int(y) >= 0 and int(y) < 256, data.split('.', 3))) == 4
      return True
   except:
      return False
```

To test the function you can invoke the following:

```
>>> isIPv4('10.2.3.4') == True
True
>>> False == isIPv4('10.2.3.a') == isIPv4('10.2.3.a.5') == isIPv4('10.2.3.-2') == isIPv4('10.2.3.403')
True
```

isMulticast

A similar test can be done for multicast addresses as follows. A multicast address, for our implementation, is an IPv4 address for which the first four most significant bits are 0111.

```
import socket, struct
def isMulticast(data):
   try:
      m, = struct.unpack('>I', socket.inet_aton(data))
      return ((m & 0xF0000000) == 0xE0000000) # class D: 224.0.0.0/4 or first four bits as 0111
   except:
      return False
```

The test can be done as follows:

```
>>> isMulticast('224.0.1.2') == True
True
>>> False == isMulticast('10.2.3.4')
True
```

Now that we have looked at various aspects of parsing and formatting SIP addresses and URIs, we can move on to the actual SIP message parsing and formatting and eventually implementation of a complete SIP stack in the next chapter.

Session Initiation Protocol (SIP)

Implementing core SIP as per RFC 3261

We have already seen how to implement the addressing module. This chapter describes the implementation of the SIP module named `rfc3261`. We continue with the parsing and formatting methods from the addressing module to the SIP message structure. After describing the parsing and formatting, we move on to building a SIP stack.

SIP message

An example SIP message is shown below.

```
INVITE sip:bob@example.net SIP/2.0
Via: SIP/2.0/UDP pc33.home.com;branch=z9hG4bKnashds8
Max-Forwards: 70
To: Bob <sip:bob@example.net>
From: Alice <sip:alice@home.com>;tag=1928301774
Call-ID: a84b4c728ca8@mypc.home.com
CSeq: 613 INVITE
Contact: <sip:alice@pc33.home.com>
Content-Type: application/sdp
Content-Length: 148

v=0
o=user1 53655765 2353687637 IN IP4 192.1.2.3
s=Weekly conference call
c=IN IP4 192.1.2.3
t=0 0
m=audio 8080 RTP/AVP 0 8
m=video 8082 RTP/AVP 31
```

The first line is a request or response line, which is followed by header lines, and finally the message body. You can identify the SIP addresses and URIs in various parts of this message such as `request-URI` on the firt line and the values of `To`, `From` and `Contact` headers.

To encapsulate a SIP message we define a class `Message`. To encapsulate individual header we define a class `Header`. We take the bottom-up approach of first parsing and formatting a `Header` and then defining various methods in a `Message`.

As with addresses, we would want dynamic attributes in `Message` object to represent the various headers. Similar the `Header` can have dynamic attributes for the parameters. Some desired operations are shown below.

```
>>> m = Message("INVITE sip:kundan@example.net SIP/2.0\r\n");
>>> m.To = Header('"Kundan Singh" <sip:kundan@example.net>', "To");
>>> print m.To.value.uri.host
example.net
```

```
>>> m.method = “MESSAGE”
>>> print m
MESSAGE sip:kundan@example.net SIP/2.0
To: “Kundan Singh” <sip:kundan@example.net>
```

Header

From the structure point of view, there are four types of SIP headers: standard, address-based, comma-included and unstructured. Most SIP headers are defined to be standard headers that have a value and zero or more parameters separated by a semi-colon. The address-based headers have the value consisting of a SIP address, but the parameters are similar to the standard headers. The difference arises because the value of an address-based header can internally have “;” whereas those are forbidden in the value of the standard header. For example, URI parameters are also separated by “;” within the value of an address-based header. A comma-included header is the one that can have a “,” in the value of the header. Normally a standard or address-based header can have multiple header values in the same header line, where the values are separated by comma “,”. However, for a comma-included header such as `WWW-Authenticate`, there can be only one value per header line, and the intermediate comma “,” are part of the value. The comma-included headers are only used because of interoperability with existing HTTP headers for authentication, which are comma-included. The unstructured headers have one value per header line and the value is treated as opaque string without any structure internally. An example is `Call-ID` header.

The specification defines parsing rules for various headers, which allow us to classify them among these categories as follows. Any header name that is not covered in the following three categories is assumed to be a standard header.

```
_address      = ['contact', 'from', 'record-route', 'refer-to', 'referred-by', 'route', 'to']
_comma        = ['authorization', 'proxy-authenticate', 'proxy-authorization', 'www-authenticate']
_unstructured = ['call-id', 'cseq', 'date', 'expires', 'max-forwards', 'organization', 'server', 'subject', 'timestamp', 'user-agent']
```

An extension to SIP can define new header names in these categories. By default we assume standard header if not found in the list above.

> From RFC3261 p.32 – SIP provides a mechanism to represent common header field names in an abbreviated form. A compact form MAY be substituted for the longer form of a header field name at any time without changing the semantics of the message. A header field name MAY appear in both long and short forms within the same message. Implementations MUST accept both the long and short forms of each header name.

Besides these categories needed for our implementation, the specification also defines the short-form of header names as follows.

```
_short        = ['allow-events', 'u', 'call-id', 'i', 'contact', 'm', 'content-encoding', 'e', 'content-length', 'l', 'content-type', 'c', 'event',
'o', 'from', 'f', 'subject', 's', 'supported', 'k', 'to', 't', 'via', 'v']
```

Canonicalize

In the above listing, we have used the lower-case header names so that comparison can be done consistently. For the purpose of canonical representation and formatting of headers, as well as comparison of two header names, the standard defines canonical representation of various header names. In particular, the header names in canonical form have one or more words joined together by a dash '-' with the first letter of each word capitalized. The following statement can convert a lower-case header name to its canonical form.

```
'-'.join([x.capitalize() for x in s.split('-')])
```

There are three exceptions to this rule as shown below.

```
_exception    = {'call-id':'Call-ID','cseq':'CSeq','www-authenticate':'WWW-Authenticate'}
```

To facilitate canonicalization of header names, we define a function that first converts the name to lower case and then canonicalizes it keeping the exceptions and short-forms in mind.

```
def _canon(s):
   s = s.lower()
   return ((len(s)==1) and s in _short and _canon(_short[_short.index(s)-1])) \
      or (s in _exception and _exception[s]) or '-'.join([x.capitalize() for x in s.split('-')])
```

The method can be tested to produce canonical representations of various header names, existing or future.

```
   >>> print _canon('call-Id')
   Call-ID
   >>> print _canon('fRoM')
   From
   >>> print  _canon('refer-to')
   Refer-To
```

Quote and unquote

Another utility functionality we need is to quote and unquote a string. The parameter values in a header can be quoted, whereas we store unquoted value in our object. The following functions allow us to quote or unquote a string as applicable.

```
_quote   = lambda s: '"' + s + '"' if s[0] != '"' != s[-1] else s
_unquote = lambda s: s[1:-1] if s[0] == '"' == s[-1] else s
```

Parsing

A SIP header contains a header name and a header value. There can be any number of header attributes. The attribute name need not be known in advance. We define our class such that `name` and `value` properties

refer to the header name and value, whereas the header object itself can be used as an associative array to extract the attribute value indexed by the attribute name, e.g., `h["tag"]` or `h.tag`.

To parse a header value, we define a method that takes the header name and based on the type it invokes different parsing logic.

```
class Header(object):
   def _parse(self, value, name):
      if name in _address:
         ... # parse as address-based header
      elif name not in _comma and name not in _unstructured:
         ... # parse as standard header
      if name in _comma:
         ... # parse as comma-included header
      return value
```

For an address-based header, the returned `value` is an object of type `Address` which stores the address part of the value. The rest of the string is parsed for sequence of header parameters separated by semi-colon ";", and stored as attributes of the local `Header` object. The `Address` is set to always use quotes for the URI while formatting. This is important to prevent missing quotes which causes the URI parameters to be treated as header parameters after formatting. Note that the parameter name is considered to be case in-sensitive.

```
      if name in _address: # address header
         addr = Address(); addr.mustQuote = True
         count = addr.parse(value)
         value, rest = addr, value[count:]
         if rest:
            for n,sep,v in map(lambda x: x.partition('='), rest.split(';') if rest else []):
               if n.strip():
                  self.__dict__[n.lower().strip()] = v.strip()
```

For a standard header, the returned value is the string up to the semi-colon ";" if any, otherwise the whole value string. If the parameters are present indicated by the semi-colon ";" then they are parsed into this `Header` object as before. TODO: we need to check if the parameter name and/or value are tokens.

```
      elif name not in _comma and name not in _unstructured: # standard
         value, sep, rest = value.partition(';')
         for n,sep,v in map(lambda x: x.partition('='), rest.split(';') if rest else []):
            self.__dict__[n.lower().strip()] = v.strip()
```

A comma included header is usually of the form "`value param1=value1, param2=value2,...`" For programming convenience, we return the `value` part as the value of the header and store the individual `param-value` pairs in the local `Header` object as an associative array.

```
      if name in _comma:
         self.authMethod, sep, rest = value.strip().partition(' ')
         for n,v in map(lambda x: x.strip().split('='), rest.split(',') if rest else []):
```

```
            self.__dict__[n.lower().strip()] = _unquote(v.strip())
```

After the parsing is completed, we may want to inspect some of the unstructured header values and store the values in a more structured form. For instance, the `CSeq` header value has two parts: the number and the method name.

```
CSeq: 1 INVITE
```

For programming convenience, we can store the individual parts separately as follows. We also canonicalizes the value so as to remove more than one spaces between the number and the method name, if needed.

```
    elif name == 'cseq':
        n, sep, self.method = map(lambda x: x.strip(), value.partition(' '))
        self.number = int(n); value = n + ' ' + self.method
```

Now that we have completed the parsing step, we can create the constructor which takes an optional string for the value. The constructor removes any extra surrounding white-spaces from the value before parsing it. The header name is converted to lower-case if applicable.

```
def __init__(self, value=None, name=None):
    self.name = name and _canon(name.strip()) or None
    self.value = self._parse(value.strip(), self.name and self.name.lower() or None)
```

Formatting

Formatting a `Header` object has two semantics: either you can format only the value or the complete header line. We implement two different methods, `str` and `repr`, to achieve these functions. Correspondingly, the object can be converted to string in different contexts differently.

The value is formatted as follows. If the header type is comma-included or unstructured, then the value property is the actual value string representation which can be returned, otherwise the parameters (or rest) needs to be appended to the value. When appending the parameters, all indices from the local associative array are used except for pre-defined indices of `name`, `value` and `_viauri`.

```
    def __str__(self):
        name = self.name.lower()
        rest = '' if ((name in _comma) or (name in _unstructured)) \
            else (';'.join(map(lambda x: self.__dict__[x] and '%s=%s'%(x.lower(),self.__dict__[x]) or x, filter(lambda x:
x.lower() not in ['name','value', '_viauri'], self.__dict__))))
        return str(self.value) + (rest and (';'+rest) or '');
```

The `repr` method just returns the header “name: value” where value is formatted using the `str` method.

```
def __repr__(self):
    return self.name + ": " + str(self)
```

Misc

Besides the parsing and formatting methods, there are other utility methods needed for a `Header` object. A `dup` method is used to clone the object by formatting and parsing back into a new object.

```
def dup(self):
    return Header(self.__str__(), self.name)
```

The parameter access can be done either by container syntax (such as h["tag"]) of attribute access syntax (h.tag). This gives more flexibility to the application developer. As noted earlier, we store the parameters in the local `__dict__` property which readily allows attribute access syntax. To add the container access syntax we add the following methods.

```
def __getitem__(self, name): return self.__dict__.get(name.lower(), None)
def __setitem__(self, name, value): self.__dict__[name.lower()] = value
def __contains__(self, name): return name.lower() in self.__dict__
```

Via URI

The `Via` header is unique, in the sense that even though it is a standard header there is lot of structure inside the value part of the header.

```
Via: SIP/2.0/UDP pc33.home.com;branch=z9hG4bKnashds8
```

The `viaUri` property represents a URI object derived from the `Via Header` object such that the URI represents the address to which we need to send a response. RFC3261 specifies the process to derive such a URI. First we separate the header value "SIP/2.0/UDP pc33.home.com" into the first and second parts. The first part gives us the type: `udp`, `tcp` or `tls`.

```
        proto, addr = self.value.split(' ')
        type = proto.split('/')[2].lower()  # udp, tcp, tls
```

The second part can be used to construct a new URI object with no "user" part, a default "transport" parameter derived from the type and the "scheme" of "sip:" The URI gets stored internally.

```
        self._viaUri = URI('sip:' + addr + ';transport=' + type)
```

A default port number of 5060 is assumed if missing in the URI.

```
        if self._viaUri.port == None: self._viaUri.port = 5060
```

If the `rport` parameter is present in the header, the URI port is changed to the `rport` value if present, and not changed if `rport` value is not present.

```
        if 'rport' in self:
            try: self._viaUri.port = int(self.rport)
            except: pass # probably not an int
```

If the `type` is not a reliable transport type, and `maddr` parameter is present then the URI host is changed to `maddr` value, otherwise if received parameter is present then URI host is changed to `received` parameter value.

```
        if type not in ['tcp','sctp','tls']:
            if 'maddr' in self: self._viaUri.host = self.maddr
            elif 'received' in self: self._viaUri.host = self.received
```

We implement this function using the `viaUri` property as follows.

```
    @property
    def viaUri(self):
        if not hasattr(self, '_viaUri'):
            if self.name != 'Via': raise ValueError, 'viaUri available only on Via header'
            proto, addr = self.value.split(' ')
            type = proto.split('/')[2].lower()  # udp, tcp, tls
            self._viaUri = URI('sip:' + addr + ';transport=' + type)
            if self._viaUri.port == None: self._viaUri.port = 5060
            if 'rport' in self:
                try: self._viaUri.port = int(self.rport)
                except: pass # probably not an int
            if type not in ['tcp','sctp','tls']:
                if 'maddr' in self: self._viaUri.host = self.maddr
                elif 'received' in self: self._viaUri.host = self.received
        return self._viaUri
```

Before continuing it may be worthwhile to test our function for correctness.

```
    >>> print Header('SIP/2.0/UDP example.net:5090;ttl=1', 'Via').viaUri
    sip:example.net:5090;transport=udp
    >>> print Header('SIP/2.0/UDP 192.1.2.3;rport=1078;received=76.17.12.18;branch=0', 'Via').viaUri
    sip:76.17.12.18:1078;transport=udp
    >>> print Header('SIP/2.0/UDP 192.1.2.3;maddr=224.0.1.75', 'Via').viaUri
    sip:224.0.1.75:5060;transport=udp
```

Splitting header line

> From RFC3261 p29 – HTTP/1.1 also specifies that multiple header fields of the same field name whose value is a comma-separated list can be combined into one header field. That applies to SIP as well, but the specific rule is different because of the different grammars. Specifically, any SIP header whose grammar is of the form
>
> header = "header-name" HCOLON header-value *(COMMA header-value)
>
> allows for combining header fields of the same name into a comma- separated list.
>
> Each header field consists of a field name followed by a colon (":") and the field value. The formal grammar for a message-header allows for an arbitrary amount of whitespace on either side of the colon.

The SIP message parser need to first divide the headers portion into individual header lines, extract the header name and value from the header line and finally invoke the `Header` constructor to construct individual header object. If a single header line contains multiple comma "," separated header values, then those individual header values need to be constructed independently.

We define a class method to perform this task. The method takes a string and returns a tuple with two values: the first is the header name, and the second is an array of `Header` objects.

```
@staticmethod
def createHeaders(value):
  name, value = map(str.strip, value.split(':', 1))
  return (_canon(name),  map(lambda x: Header(x, name), value.split(',') if name.lower() not in _comma else [value]))
```

This can be tested as follows.

```
>>> print Header.createHeaders('Event: presence, reg')
('Event', [Event: presence, Event: reg])
```

Now that we have implemented the class, we can do basic testing as follows:

```
>>> print repr(Header('"Kundan Singh" <sip:kundan@example.net>', 'To'))
To: "Kundan Singh" <sip:kundan@example.net>
>>> print repr(Header('"Kundan"<sip:kundan99@example.net>', 'To'))
To: "Kundan" <sip:kundan99@example.net>
>>> print repr(Header('Sanjay <sip:sanjayc77@example.net>', 'fRoM'))
From: "Sanjay" <sip:sanjayc77@example.net>
>>> print repr(Header('application/sdp', 'conTenT-tyPe'))
Content-Type: application/sdp
>>> print repr(Header('presence; param=value;param2=another', 'Event'))
Event: presence;param=value;param2=another
>>> print repr(Header('78  INVITE', 'CSeq'))
CSeq: 78 INVITE
```

Message

A `Message` object is a representation of a SIP message. Unlike other SIP stacks that define various individual message types, separate classes for first line, etc., in Python we use dynamic properties to easily implement those features. In particular, the same class implements both the request and response. The attributes such as `method` or `response`.are valid for request or response, respectively.

Even though we would like to have the attribute syntax for accessing the header from a message, certain header names cannot be a Python attribute name. For example, "Content-Length" with an embedded dash cannot be an attribute. Thus, we implement both the attribute as well as container access for the headers in a message. The header names are case in-sensitive. Accessing the header that doesn't exist in the message gives `None` instead of exception. This creates cleaner source code, instead of having to catch exceptions everywhere. The following definition allows us to create a generic object that can hold name value pairs and allow access using both attribute access and container access syntax.

```
class Message(object):
    def __getattr__(self, name):          return self.__getitem__(name)
    def __getattribute__(self, name):     return object.__getattribute__(self, name.lower())
    def __setattr__(self, name, value):   object.__setattr__(self, name.lower(), value)
    def __delattr__(self, name):          object.__delattr__(self, name.lower())
    def __hasattr__(self, name):          object.__hasattr__(self, name.lower())
    def __getitem__(self, name):          return self.__dict__.get(name.lower(), None)
    def __setitem__(self, name, value): self.__dict__[name.lower()] = value
    def __contains__(self, name):         return name.lower() in self.__dict__
```

There are certain pre-defined attributes: `method`, `uri`, `response`, `responsetext`, `protocol` and `body`.

```
    _keywords = ['method','uri','response','responsetext','protocol','_body','body']
```

A SIP header can be either a single-instance header or a multiple-instance header. There are only a few single-instance headers. By default, a header is treated as multiple-instance. The difference between single and multiple instance headers is that we can expose a single `Header` as the value of a single instance header, whereas we expose an `list` of `Header` objects as the value of a multiple instance header.

```
    _single = ['call-id', 'content-disposition', 'content-length', 'content-type', 'cseq', 'date', 'expires', 'event', 'max-forwards',
'organization', 'refer-to', 'referred-by', 'server', 'session-expires', 'subject', 'timestamp', 'to', 'user-agent']
```

Parsing

> From RFC3261 p.27 – The start-line, each message-header line, and the empty line MUST be terminated by a carriage-return line-feed sequence (CRLF). Note that the empty line MUST be present even if the message-body is not.

Parsing a SIP message is an important method in the Message class. The first step in parsing a SIP message is splitting the string across "\r\n\r\n" so that the second part becomes the message body text, and the first part contains the first line as well as the headers.

```
def _parse(self, value):
   firstheaders, body = value.split('\r\n\r\n', 1)
```

From RFC3261 p.28 – SIP requests are distinguished by having a Request-Line for a start-line. A Request-Line contains a method name, a Request-URI, and the protocol version separated by a single space (SP) character.

Request-Line = Method SP Request-URI SP SIP-Version CRLF

SIP responses are distinguished from requests by having a Status-Line as their start-line. A Status-Line consists of the protocol version followed by a numeric Status-Code and its associated textual phrase, with each element separated by a single SP character.

Status-Line = SIP-Version SP Status-Code SP Reason-Phrase CRLF

After splitting the message string into two parts, the first part is further split into the first line and the headers text.

```
   firstline, headers = firstheaders.split('\r\n', 1)
```

The first line can be either a request line or a response line. This can be identified by splitting the first line into three parts across a white-space character and checking if the second partition is an integer or not? If it is an integer (i.e., a response code), then the first line is a response line, otherwise it is a request line. The properties `response` (of type `int`), `responsetext` and `protocol` are set for a response line and the properties `method`, `uri` (or type URI) and `protocol` are set for a request line.

```
   a, b, c = firstline.split(' ', 2)
   try:   # try as response
      self.response, self.responsetext, self.protocol = int(b), c, a # throws error if b is not int.
   except: # probably a request
      self.method, self.uri, self.protocol = a, URI(b), c
```

SIP header fields are similar to HTTP header fields in both syntax and semantics. In particular, SIP header fields follow the HTTP definitions of syntax for the message-header and the rules for extending header fields over multiple lines. However, the latter is specified in HTTP with implicit whitespace and folding. This specification conforms to RFC 2234 and uses only explicit whitespace and folding as an integral part of the grammar.

After the first line is parsed, the headers text is split into individual header lines. Note that a header line that starts with a white-space character is a continuation of the previous header line. Let's not worry about the continuation line for now.

```
   for h in headers.split('\r\n'):
      if h.startswith(r'[ \t]'):
         pass
      ... # parse the header line
```

To parse the header line we use the `createHeaders` class method in the `Header` class, which returns a tuple with two elements: the header name and the array of `Header` objects indicating individual header values. The header values are stored in the local object indexed by the header name, with value as either a single `Header` object or a list of `Header` objects. Any error while parsing the header line is ignored and we continue to the next header line.

```
        try:
            name, values = Header.createHeaders(h)
            if name not in self: # doesn't already exist
                self[name] = values if len(values) > 1 else values[0]
            elif name not in Message._single: # valid multiple-instance header
                if not isinstance(self[name],list): self[name] = [self[name]]
                self[name] += values
        except:
            continue
```

> From RFC3261 p.33 – Requests, including new requests defined in extensions to this specification, MAY contain message bodies unless otherwise noted. The interpretation of the body depends on the request method.
>
> For response messages, the request method and the response status code determine the type and interpretation of any message body. All responses MAY include a body.
>
> The body length in bytes is provided by the Content-Length header field.

Once we have parsed the headers text into individual header elements, we extract the message body. SIP defines a `Content-Length` of 0 if that header is missing. Once the body is stored in the `body` property, we validate the body length and throw an exception if there is a mismatch.

```
        bodyLen = int(self['Content-Length'].value) if 'Content-Length' in self else 0
        if body: self.body = body
        if self.body != None and bodyLen != len(body):
            raise ValueError, 'Invalid content-length %d!=%d'%(bodyLen, len(body))
```

As the last step in the parsing process, we check if the mandatory headers are present or not.

```
        for h in ['To','From','CSeq','Call-ID']:
            if h not in self: raise ValueError, 'Mandatory header %s missing'%(h)
```

There are a number of boundary conditions that we need to implement, but haven't implemented so far. Examples are (1) if the message doesn't contain "\r\n\r\n" sequence than the message body should be assumed to be empty, (2) should parse as a valid message even if there are no headers, because the application can add headers later, (3) should throw an error if the first line has less than three parts, (4) should validate the syntax of the `protocol` property, (5) the first header should not start with a white-space, (6) the `method` and `response` properties should be validated, (7) the syntax of top-most `Via` header and fields such as `ttl`, `maddr`, `received` and `branch` should be validated.

Once we have implemented the parsing method, we can build the constructor that takes the optional message string to parse.

```
    def __init__(self, value=None):
        self.method = self.uri = self.response = self.responsetext = self.protocol = self._body = None
        if value: self._parse(value)
```

Formatting

The formatting of the SIP message is simpler than parsing. The idea is to construct the first line followed by individual header lines and finally append the message body. The `Message` object allows iteration over the associative array index, where the iteration walks over all the `Header` objects.

```
def __repr__(self):
    if self.method != None: m = self.method + ' ' + str(self.uri) + ' ' + self.protocol + '\r\n'
    elif self.response != None: m = self.protocol + ' ' + str(self.response) + ' ' + self.responsetext + '\r\n'
    else: return None # invalid message
    for h in self:
        m += repr(h) + '\r\n'
    m+= '\r\n'
    if self.body != None: m += self.body
    return m
```

Cloning

Cloning a message is similar to earlier data structures – format to string and parse the string back into another `Message` object.

```
def dup(self):
    return Message(self.__repr__())
```

Accessing headers

As mentioned earlier the attribute and container access can be used to refer to or add a particular header. However, there are some additional convenient methods that we would want to implement to access the headers. The iteration over the object should return each header in turn. This is implemented by flattening the headers into a single list, and returning the iterator on that list.

```
def __iter__(self):
    h = list()
    for n in filter(lambda x: not x.startswith('_') and x not in Message._keywords, self.__dict__):
        h += filter(lambda x: isinstance(x, Header), self[n] if isinstance(self[n],list) else [self[n]])
    return iter(h)
```

The method `first` returns the first occurrence of a particular `Header` object from the header name. If the header doesn't exist then it returns `None`. This method can be used when a header object is needed in a singular context, irrespective of whether the header is a single or multiple instance header.

```
def first(self, name):
    result = self[name]
```

```
    return isinstance(result,list) and result[0] or result
```

The method `all` returns a list of all the Header objects from the given header name. Event if the header type is single-instance, it returns a list containing single element. This method is useful when accessing the header object in a list context irrespective of whether the header is a single or multiple instance header. The method is further extended to accept a list of header names and return all the `Header` objects associated with all those names. Thus, `h.all("To","From","CSeq","Call-ID")` will return a list of all those mandatory headers. If no such headers are found, then it returns an empty list, instead of `None`. Thus the return value can always be evaluated in a list context.

```
def all(self, *args):
    args = map(lambda x: x.lower(), args)
    h = list()
    for n in filter(lambda x: x in args and not x.startswith('_') and x not in Message._keywords, self.__dict__):
        h += filter(lambda x: isinstance(x, Header), self[n] if isinstance(self[n],list) else [self[n]])
    return h
```

The method `insert` can be used to insert a particular `Header` in a `Message`. The application doesn't have to worry about whether it is a single or multiple instance header and how many occurrences exist in the message. An optional flag allows appending the header instead of inserting at the beginning. TODO: we should not insert multiple instance header if the header name indicates a single instance header type.

```
def insert(self, header, append=False):
    if header and header.name:
        if header.name not in self:
            self[header.name] = header
        elif isinstance(self[header.name], Header):
            self[header.name] = (append and [self[header.name], header] or [header, self[header.name]])
        else:
            if append: self[header.name].append(header)
            else: self[header.name].insert(0, header)
```

Accessing message body

We implement a read-write property named `body`, which refers to the message body. When the `body` is explicitly set, we also update the `Content-Length` header value so that the message's content length remains consistent.

```
def body():
    def fset(self, value):
        self._body = value
        self['Content-Length'] = Header('%d'%(value and len(value) or 0), 'Content-Length')
    def fget(self):
        return self._body
    return locals()
body = property(**body())
```

Accessing response type

We implement various read-only properties, `is1xx`, `is2xx`, etc., to indicate whether a `Message` object represents a response of that particular response class. Finally, a `isfinal` property indicates whether the message is a final response or not. Python allows us to dynamically create methods and properties as follows.

```
for x in range(1,7):
   exec 'def is%dxx(self): return self.response and (self.response / 100 == %d)'%(x,x)
   exec 'is%dxx = property(is%dxx)'%(x,x)
@property
def isfinal(self): return self.response and (self.response >= 200)
```

Creating a request or response

Instead of having the application create the `Message` object and populate the fields, it would be better to define the factory methods to create different types of messages. We implement two class methods, `createRequest` and `createResponse`, that can be used by the application to create a request or response `Message`, respective, by supplying appropriate parameters. The use of these methods ensures that the created object will be valid. For example, the `uri` property is actually a URI for a request, and the `protocol` property actually stores "SIP/2.0".

Before defining those methods, let's define a `populateMessage` method that updates the `Message` object with the supplied `headers` and message body `content`. If no message body is specified, then it resets the `Content-Length` header value.

```
@staticmethod
def _populateMessage(m, headers=None, content=None):
   if headers:
      for h in headers: m.insert(h, True) # append the header instead of overriding
   if content: m.body = content
   else: m['Content-Length'] = Header('0', 'Content-Length')
```

Now to create a request, we create the `Message` object, and populate the `method`, `uri`, `protocol` properties. Then the optional headers and message body are populated. Finally, the `CSeq` header, if present, is updated with the correct `method` name. This allows us to create a new request from the headers of an existing request, and let this method update the headers accordingly.

```
@staticmethod
def createRequest(method, uri, headers=None, content=None):
   m = Message()
   m.method, m.uri, m.protocol = method, URI(uri), 'SIP/2.0'
   Message._populateMessage(m, headers, content)
   if m.CSeq != None and m.CSeq.method != method: m.CSeq = Header(str(m.CSeq.number) + ' ' + method, 'CSeq')
   return m
```

A response can be created by supplying various properties. Optionally, the original request can be supplied as well. If the original request is provided, then the response uses the `To`, `From`, `CSeq`, `Call-ID` and `Via` headers from the original request. As per RFC3261, if the response code is 100, then the `Timestamp` header is also copied from the original request. If optional headers are provided, then those are used to override the previously assigned headers if needed.

> The From field of the response MUST equal the From header field of the request. The Call-ID header field of the response MUST equal the Call-ID header field of the request. The CSeq header field of the response MUST equal the CSeq field of the request. The Via header field values in the response MUST equal the Via header field values in the request and MUST maintain the same ordering.
>
> If a request contained a To tag in the request, the To header field in the response MUST equal that of the request. However, if the To header field in the request did not contain a tag, the URI in the To header field in the response MUST equal the URI in the To header field.

```
@staticmethod
def createResponse(response, responsetext, headers=None, content=None, r=None):
    m = Message()
    m.response, m.responsetext, m.protocol = response, responsetext, 'SIP/2.0'
    if r:
        m.To, m.From, m.CSeq, m['Call-ID'], m.Via = r.To, r.From, r.CSeq, r['Call-ID'], r.Via
        if response == 100: m.Timestamp = r.Timestamp
    Message._populateMessage(m, headers, content)
    return m
```

At this point, we have seen how to parse and format a SIP message and how to define various data structures for easy access of the properties in a message or its header. In the next section, we detail the implementation of a SIP stack, including various layers as per the specification.

SIP stack

Although there is no good definition of a SIP stack, we use the following block diagram to decompose the core SIP implementation components which we refer to as SIP stack. As shown in the diagram, a SIP stack consists of these components: UserAgent/Dialog, Transaction, Message and Transport. The Transport, Transaction and UserAgent/Dialog layers are defined in RFC3261.

> From RFC3261 p.18 – SIP is structured as a layered protocol, which means that its behavior is described in terms of a set of fairly independent processing stages with only a loose coupling between each stage.
>
> Not every element specified by the protocol contains every layer. Furthermore, the elements specified by SIP are logical elements, not physical ones. A physical realization can choose to act as different logical elements, perhaps even on a transaction-by-transaction basis. The lowest layer of SIP is its syntax and encoding. Its encoding is specified using an augmented Backus-Naur Form grammar (BNF).
>
> The second layer is the transport layer. It defines how a client sends requests and receives responses and how a server receives requests and sends responses over the network. All SIP elements contain a transport layer.

The Message layer defines the SIP message parsing and formatting as per the specification, and as we saw in the previous section. Although the specification keeps the transport layer above the syntax and encoding layer, we keep the implementation of the syntax and encoding layer (in the form of Message layer) above the transport layer. This is needed for implementations which treat the transport as the socket layer of the operating system with some methods to perform SIP related functions. This also helps us in moving the

transport layer to an external entity such that the SIP implementation becomes independent of the actual transport layer.

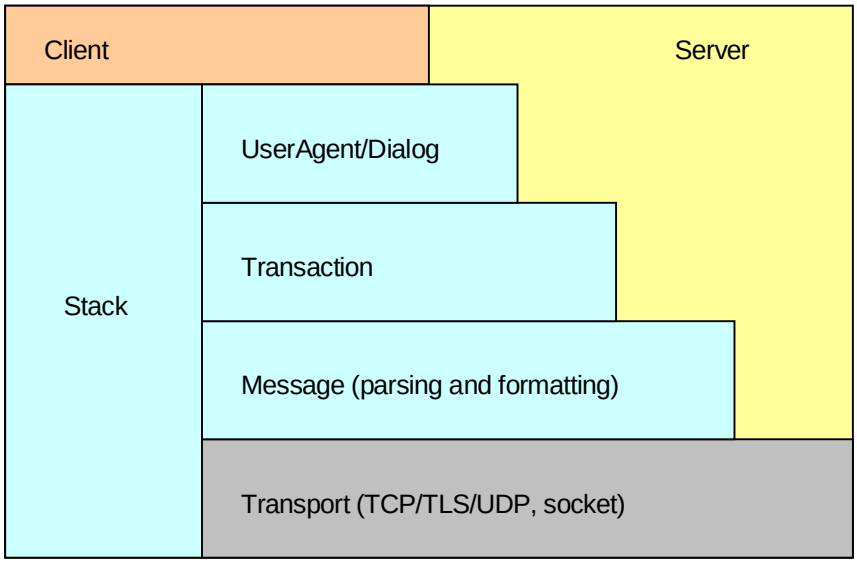

Fig. block diagram of an example SIP stack

The third layer is the transaction layer. Transactions are a fundamental component of SIP. A transaction is a request sent by a client transaction (using the transport layer) to a server transaction, along with all responses to that request sent from the server transaction back to the client. The transaction layer handles application-layer retransmissions, matching of responses to requests, and application-layer timeouts. Any task that a user agent client (UAC) accomplishes takes place using a series of transactions. User agents contain a transaction layer, as do stateful proxies.

The layer above the transaction layer is called the transaction user (TU). Each of the SIP entities, except the stateless proxy, is a transaction user. When a TU wishes to send a request, it creates a client transaction instance and passes it the request along with the destination IP address, port, and transport to which to send the request. A TU that creates a client transaction can also cancel it. When a client cancels a transaction, it requests that the server stop further processing, revert to the state that existed before the transaction was initiated, and generate a specific error response to that transaction.

Certain other requests are sent within a dialog. A dialog is a peer-to-peer SIP relationship between two user agents that persists for some time. The dialog facilitates sequencing of messages and proper routing of requests between the user agents.

A SIP client can be built on top of the UserAgent/Dialog layer whereas a SIP server can be built at various layers depending on the features in the server – load-balancing server, transaction stateless proxy, transaction stateful proxy, registrar, call stateful server.

The Stack layer represents the general processing module that needs to interact with all the other layers. We describe the individual layers in detail below.

Stack and Transport

For the purpose of this implementation, we assume that the actual transport is outside our module, `rfc3261`. This allows us to implement the core SIP functions without worrying about the network transport layer. As a side-effect, we do not have to worry about the process model, whether it is event-based or thread-pool, because the transport layer usually controls the messaging and architecture. This step is tricky and it is important that you pay attention to the details here to understand the implementation.

The Stack later is the main interface for our SIP implementation. In a SIP implementation, typically we listen on a transport address for incoming packet. When a packet is received, it is parsed and depending on the message it gets delivered to either the transaction, user-agent or dialog layer. The Stack module receives a message from the external transport and delivers it appropriately. The individual modules such as transaction, user-agent and dialog can have their own timers. When these timers expire the module takes certain actions, such as retransmitting a response or a request. We again use the Stack module to deliver messages to be sent to the transport. When the application wants to send a request or a response, such as SIP registration or call answer, it uses the Stack module to initiate the outbound request or response processing. Eventually, the Stack layer delivers it back to the external transport layer for actual transport of the message. Thus, the Stack layer is the sole interface in and out of our SIP implementation.

TransportInfo

The application may listen on multiple transport addresses, e.g., one for UDP, and one for TCP, or multiple UDP ports. We simplify our design by assuming that each instance of the `Stack` object is associated with a single instance of the listening transport. The `Stack` needs some information about the associated transport for processing various SIP functions. This information can be encapsulated in an object and supplied to the Stack on construction. An example of such information object is defined below.

```
class TransportInfo:
   def __init__(self, sock, secure=False):
      addr = getlocaladdr(sock)
      self.host, self.port = addr[0], addr[1]
      self.type = (sock.type==socket.SOCK_DGRAM and 'udp' or 'tcp')
      self.secure = secure
      self.reliable = self.congestionControlled = (sock.type==socket.SOCK_STREAM)
```

We assume that the external function `getlocalsock` returns the `(ip, port)` tuple for the locally bound socket `sock`. The actual details of this data object is not important, but what is important is that the data object should hold these properties: `host` as the dotted local IP address, `port` as the listening port number, `type` as one of “udp”, “tcp”, “tls” or “sctp” to indicate the transport type, `secure` as Boolean indicating whether the transport is secure or not, and `reliable` and `congestionControlled` as Booleans indicating whether the transport is reliable and congestion controlled, respectively, or not.

Such a data object is supplied to the constructor of the `Stack` object. The application also needs to install itself as a listener of the events from the `Stack` object, so that it can know about incoming call or successful call events, etc.

```
class Stack(object):
   def __init__(self, app, transport):
      … # construct a Stack
```

Application Callback Interface

Here, the `transport` argument is of type `TransportInfo` or something similar, and the `app` argument is a reference to the application. The `Stack` object invokes various methods on the `app` object. In particular the `app` object must implement several interface methods: `send`, `sending`, `createServer`, `receivedRequest`, `receivedResponse`, `cancelled`, `dialogCreated`, `authenticate` and `createTimer`. All these interface methods take the last argument as a reference to the `Stack` object that is calling the method. This allows the application to use multiple stacks, e.g., one for UDP and another for TCP transport.

To send some data on the transport to some destination, the `Stack` object calls the `app.send` method with first parameter as the data string to be sent and the second parameter as a host-port tuple, e.g., ('192.1.2.3', 5060). Thus the application must implement the following method.

```
def send(self, data, addr, stack):
    'send data (str) to addr of form ('192.1.2.3', 5060).'
```

When the `Stack` receives an incoming request and needs to create a UAS (user agent server), it invokes the `app.createServer` method with first argument as the request `Message` and second as the `URI` from the request line. The application must implement the method and return either a valid `UserAgent` object if it knows how to handle this incoming request, else `None` if it does not know how to handle this incoming request. For example, a client implementation will typically return `None` for a REGISTER request.

```
def createServer(self, request, uri, stack):
    return UserAgent(stack, request) if request.method != "REGISTER" else None
```

The `Stack` invokes `receivedRequest` and `receivedResponse` methods on `app` to indicate incoming request or response associated with a `UserAgent`. The first argument is the `UserAgent` object reference, and the second is the `Message` representing the request or response.

```
def receivedRequest(self, ua, request, stack): ...
def receivedResponse(self, ua, response, stack): ...
```

The Stack invokes the `app.sending` method to indicate that a message is about to be sent on a `UserAgent`. This method gets invoked before doing any DNS resolution for destination address, whereas the `app.send` gets invoked to actually send a formatted message string to the destination address. The `sending` method gives an opportunity to the application to inspect and modify the `Message` if needed before it is sent out.

```
def sending(self, ua, message, stack): ...
```

If an incoming request, typically INVITE, is cancelled by remote endpoint and the `Stack` receives a CANCEL request, then it invokes `app.cancelled` instead of `app.receivedRequest`. The second argument is the original request `Message` which was cancelled. This allows the `Stack` to handle the CANCEL internally, while still inform the application about the cancellation of the original request.

```
def cancelled(self, ua, request, stack): ...
```

Sometimes the `Stack` needs to convert an existing UAS or UAC to a SIP dialog, e.g., while sending or receiving 2xx response to INVITE it creates a `Dialog` out of `UserAgent`. The application might have stored a reference to the original `UserAgent` object. The stack invokes the `app.dialogCreated` method to inform the application that a new dialog has been created from the previous UAC or UAS, and the application can then update its reference.

```
def dialogCreated(self, dialog, ua, stack): ...
```

In our implementation the `Dialog` class is derived from the `UserAgent` class so that it can reuse a number of member variables and certain methods.

When the `Stack` receives a 401 or 407 response for an outbound request, it tries to authenticate and retry the request. To do so, it needs the credentials from the application. It invokes the `app.authenticate` method to get the credentials from the application. The application should populate the authentication `username` and `password` properties in the second argument, which is an `object`. If the credentials are known and populated in the object, the application returns `True`, otherwise it returns `False`.

```
def authenticate(self, ua, obj, stack):
    obj.username, obj.password = "kundan", "mysecret"
    return True
```

Finally, the last interface method is for creating a timer. Since the core SIP implementation is independent of the thread or event model, whereas the timers in the SIP state machines require the knowledge of the model, we tried to remove this dependency by using the interface method, `app.createTimer`.

```
def createTimer(self, app, stack):
    return Timer(app)
```

This method must return a valid application defined `Timer` object. An example `Timer` class is pseudo-coded below. The constructor takes an object, on which the timer callback `timedout` is invoked. The Timer object provides a `start([delay])` and `stop()` methods to control the timer. The `delay` property stores the delay supplied in the last call to `start` so that subsequent calls without an argument can reuse the previous value. The `delay` is supplied in milliseconds.

```
class Timer(object):
    def __init__(self, app): self.delay=0; self.app = app; ...# will invoke app.timedout(self) on timeout
    def start(self, delay=None):  ... # start the timer
    def stop(self): ... # stop the timer
```

Reason for design choice

While going through this section you might have felt that the interface is very complex. Please trust me on this – given the requirement of keeping the thread-event model outside the SIP implementation, this is a very

clean and well documented interface. There is some complexity because of the added flexibility requirement. Keeping the thread-event model outside the SIP implementation allows us to implement and experiment with various threading models for performance evaluation. Moreover, my source code has example client which implements these interface methods along with the `Timer` class.

Application method interface

Now that we have described the application interface from `Stack` to the application, let's look at the methods exposed by the `Stack` which can be invoked by the application. A simple application will typically need to create a `Stack` object and invoke the `received` method whenever any data is available on the associated transport for this stack. Occasionally the application may need to access the `URI` representing the listening point for this Stack, so that it can construct other addresses, e.g., `Contact` header. Note that you must create a new URI if needed instead of modifying the `uri` property.

```
stack = Stack(self, TransportInfo(sock))
…
stack.received(dataStr, (remoteHost, remotePort))
…
c = Header(str(stack.uri), 'Contact')
c.value.uri.user = "kundan"
```

Constructing and destroying the Stack

Let's define the `Stack` class. We maintain several properties in the `Stack`. Each stack has a list of SIP methods, `serverMethods`, that are supported on the incoming direction. The `tag` property stores a unique tag value that gets added in various To and From headers. The stack also maintains two tables: one for all the `Transaction`s and other for all the `Dialog`s, which have been created.

As mentioned before the constructor takes a reference to the application to invoke the callback and a reference to the transport information object

```
class Stack(object):
    def __init__(self, app, transport):
        self.tag = str(random.randint(0,2**31))
        self.app, self.transport = app, transport
        self.closing = False
        self.dialogs, self.transactions = dict(), dict()
        self.serverMethods = ['INVITE','BYE','MESSAGE','SUBSCRIBE','NOTIFY']
```

The destructor is analogous which cleans up all the references to dialogs and transaction. We keep an internal property named `closing` to know whether the stack is being closed. Certain operations are not done on a stack that is closing, e.g., a new send request should be ignored.

```
    def __del__(self):
        self.closing = True
        for d in self.dialogs: del self.dialogs[d]
        for t in self.transactions: del self.transactions[t]
```

```
        del self.dialogs; del self.transactions
```

Listening URI, new Call-ID and Via header

The `uri` property represents the local listening transport. The URI scheme is "sips:" for secure transport and "sip:" for non-secure. The host and port portions are derived from the original transport information supplied in the constructor.

```
    @property
    def uri(self):
        transport = self.transport
        return URI(((transport.type == 'tls') and 'sips' or 'sip') + ':' + transport.host + ':' + str(transport.port))
```

The stack also provides an internal method to create a new `Call-ID` based on some random number and local transport host name. Usually the application doesn't need to use this method. This is used by other modules in the stack implementation to create a new call identifier.

```
    @property
    def newCallId(self):
        return str(random.randint(0,2**31)) + '@' + (self.transport.host or 'localhost')
```

Similarly, the stack has an internal method to create a new `Header` object representing the `Via` header. In particular it uses the transport information to populate the SIP transport type, listening host name and port number. The `rport` header parameter is put without any value.

> From RFC3261 p.39 – When the UAC creates a request, it MUST insert a Via into that request. The protocol name and protocol version in the header field MUST be SIP and 2.0, respectively.

```
    def createVia(self, secure=False):
        if not self.transport: raise ValueError, 'No transport in stack'
        if secure and not self.transport.secure: raise ValueError, 'Cannot find a secure transport'
        return Header('SIP/2.0/' +self.transport.type.upper()+' '+self.transport.host + ':' + str(self.transport.port) + ';rport', 'Via')
```

Sending and receiving on transport

To send a message to the transport via this `Stack`, the other modules invoke the `send` method with the data. The destination address is supplied if available, but may be derived from the message itself if needed. If the destination address is supplied, it must be either a URI or a host-port tuple. If it is a URI, then the host-port tuple is derived from the host and port portion of the URI. If port number is missing, then default SIP port number is used. If the data to be sent is a `Message` object, then it is formatted into a string. Once we have the data string and destination host-port tuple, we can invoke the application's `send` method to actually send the data using the associated transport. TODO: why is transport supplied?

```
    def send(self, data, dest=None, transport=None):
```

```
    if dest and isinstance(dest, URI):
        if not uri.host: raise ValueError, 'No host in destination uri'
        dest = (dest.host, dest.port or self.transport.type == 'tls' and self.transport.secure and 5061 or 5060)
        … # additional processing on Message if needed
    self.app.send(str(data), dest, stack=self)
```

We need to do some additional processing on the SIP message based on the specification before it is sent out.

> From RFC3261 p.143 – A client that sends a request to a multicast address MUST add the "maddr" parameter to its Via header field value containing the destination multicast address, and for IPv4, SHOULD add the "ttl" parameter with a value of 1.

Secondly, if a response needs to be sent and no destination address is supplied, then we use the `viaUri` property of the top-most `Via` header of the response, to decide where to send the response. Please see the description on how `viaUri` property is generated earlier in this chapter.

```
    if isinstance(data, Message):
        if data.method:     # request
            if dest and isMulticast(dest[0]):
                data.first('Via')['maddr'], data.first('Via')['ttl'] = dest[0], 1
        elif data.response: # response
            if not dest:
                dest = data.first('Via').viaUri.hostPort
```

When the application receives a message on the transport, it must invoke the `received` method on the stack to supply the received message to the stack. The source address in the form of host-port tuple is also supplied by the application. This is the only method that the application needs to invoke on the incoming direction of a message.

In this method the stack parses the received data string into a SIP message. The message is then handed over to different methods for handling a request or a response. TODO: we need to send a 400 response if there is parsing error for a non-ACK request, and response can be sent.

```
def received(self, data, src):
    try:
        m = Message(data)
        uri = URI((self.transport.secure and 'sips' or 'sip') + ':' + str(src[0]) + ':' + str(src[1]))
        if m.method:
            … # additional checks on request
            self._receivedRequest(m, uri)
        elif m.response:
            self._receivedResponse(m, uri)
        else: raise ValueError, 'Received invalid message'
    except ValueError, E:
        if _debug: print 'Error in received message:', E
```

For a received request, there are some additional checks that need to be done. In particular, it needs to check if a `Via` header exists, and if the top-most `Via` header is different from the source address, then it needs to update the top-most `Via` header with the `received` and `rport` attributes correctly.

```
if m.Via == None: raise ValueError, 'No Via header in request'
via = m.first('Via')
if via.viaUri.host != src[0] or via.viaUri.port != src[1]:
   via['received'], via.viaUri.host = src[0], src[0]
if 'rport' in via: via['rport'] = src[1]
via.viaUri.port = src[1]
```

Request processing

When an incoming request is received, we check if a matching transaction exists or not for this request. This is done by invoking the `findTransaction` method with the transaction identifier derived from the `branch` parameter of the top-most `Via` header, and optionally the request `method`. Usually the transaction identifier for CANCEL and ACK are different than the original INVITE. Hence we need the request method to distinguish between the two cases.

One special case is when the request method is ACK and the `branch` parameter is "0". Some existing implementation, such as "iptel.org" service, always puts a `branch` parameter of "0" in the ACK. Thus if there are multiple previous transaction's ACK, the new request will match the previous transaction's ACK. Either we need to fix the code to handle end-to-end ACK correctly in `findTransaction`, or we can do a work-around of not matching an ACK with branch as "0" to a transaction.

```
def _receivedRequest(self, r, uri):
   branch = r.first('Via').branch
   if r.method == 'ACK' and branch == '0':
      t = None
   else:
      t = self.findTransaction(Transaction.createId(branch, r.method))
```

If a transaction is found, the request is delivered to the transaction object for further processing, and the stack module doesn't need to care about this request anymore.

```
if not t:
   ... # processing when transaction not found
else:
   t.receivedRequest(r)
```

If no matching transaction is found, then a new server transaction needs to be created to handle the request. The creation of server transaction can be done by the dialog layer if the request is associated with an existing dialog, or by the `UserAgent` itself. If a new server transaction cannot be created for some reason, it sends a "404 Not Found" response to the source via the transport layer if the request is not ACK.

```
if not t:
   app = None # object through which new transaction is created
```

```
if r.method != 'CANCEL' and 'tag' in r.To:
  … # process an in-dialog request
elif r.method != 'CANCEL':
  … # process out-of-dialog request
else:
  … # process a CANCEL request
if app:
  t = Transaction.createServer(self, app, r, self.transport, self.tag)
elif r.method != 'ACK':
  self.send(Message.createResponse(404, "Not found", None, None, r))
```

If the request is a CANCEL request, then the original INVITE transaction is searched for the `branch` parameter. If an original transaction is found, then the new server transaction is created out of the user of this original transaction object, which could be a `UserAgent` or `Dialog` object associated with that original transaction. If no original transaction is found, then appropriate response is returned via the transport layer.

```
o = self.findTransaction(Transaction.createId(r.first('Via').branch, 'INVITE')) # original transaction
if not o:
  self.send(Message.createResponse(481, "Original transaction does not exist", None, None, r))
  return
else:
  app = o.app
```

If the request is not CANCEL and a `tag` parameter is present in the `To` header indicating that this request belongs to an existing dialog, then we search for an existing matching dialog. If a matching dialog is not found, then "481 Dialog does not exist" response is returned using the transport layer for non-ACK requests. For an ACK request if a matching dialog is not found, then we try to locate the original INVITE transaction. If a transaction is found, then the new request is delivered to that original transaction, otherwise we ignore the ACK request. No server transaction is created for an ACK request. TODO: check if this is the right processing in this case. If a matching dialog is found, then the new server transaction is created using that dialog object.

```
d = self.findDialog(r)
if not d: # no dialog found
  if r.method != 'ACK':
    self.send(Message.createResponse(481, 'Dialog does not exist', None, None, r))
  else: # ACK
    if not t and branch != '0': t = self.findTransaction(Transaction.createId(branch, 'INVITE'))
    if t: t.receivedRequest(r)
    else: print 'No existing transaction for ACK'
    return
else: # dialog found
  app = d
```

If the request is not CANCEL and there is no `tag` parameter in the `To` header, which means this is an out-of-dialog request, then the processing is as follows. The stack invokes the application's callback to create a new UAS, i.e., `UserAgent` object in server mode. If the application accepts the request and creates a `UserAgent` object then the new server transaction is created out of this `UserAgent` object, otherwise a "405 Method not allowed" response is returned for non-ACK requests via the transport layer.

```
            u = self.createServer(r, uri)
            if u:
                app = u
            elif r.method == 'OPTIONS':
                ... # handle OPTIONS separately
            elif r.method != 'ACK':
                self.send(Message.createResponse(405, 'Method not allowed', None, None, r))
                return
```

The `Stack` should respond to an out-of-dialog OPTIONS request event if the application doesn't want to create UAS. We do this by creating a "200 OK" response with the `Allow` header containing list of supported methods, but no message body – since the stack doesn't know about the session description.

```
            elif r.method == 'OPTIONS':
                m = Message.createResponse(200, 'OK', None, None, r)
                m.Allow = Header('INVITE, ACK, CANCEL, BYE, OPTIONS', 'Allow')
                self.send(m)
                return
```

Response processing

If the incoming message is parsed into a response, the processing is as follows. If the `Via` header is missing, it generates an error. Otherwise it extracts the `branch` parameter from the top-most `Via` header, and the `method` attribute from the `CSeq` header. These properties are used to create a transaction identifier to match against all existing transactions.

```
    def _receivedResponse(self, r, uri):
        if not r.Via: raise ValueError, 'No Via header in received response'
        branch = r.first('Via').branch
        method = r.CSeq.method
        t = self.findTransaction(Transaction.createId(branch, method))
```

If a matching transaction is found for the response, then the response is handed over to the transaction object for further processing.

```
        if not t:
            ... # transaction not found
        else:
            t.receivedResponse(r)
```

If no matching transaction is found for the response, then the processing depends on the response and original request type. If the response is a 2xx-class response of an INVITE request, then we try to find a matching dialog for the response. If a matching dialog is found, the response is handed over to the `Dialog` object for further processing. If no matching dialog is found, it generates an error. Similarly, for all other responses it generates an error if no matching transaction is found.

```
        if method == 'INVITE' and r.is2xx: # success of INVITE
            d = self.findDialog(r)
            if not d: raise ValueError, 'No transaction or dialog for 2xx of INVITE'
            else: d.receivedResponse(None, r)
        else: raise ValueError, 'No transaction for response'
```

Searching dialog and transaction

As mentioned before, the Stack object maintains a table of active `Dialog` and `Transaction` objects. The `findDialog` method locates an existing dialog either by a dialog identifier string or using a `Message` object. The `Dialog.extractId` method is used to extract a dialog identifier string from a `Message` object. As we will see later, a dialog identifier consists of the `Call-Id`, `local-tag` and `remote-tag` properties. The method returns `None` if a dialog is not found.

```
    def findDialog(self, arg):
        return self.dialogs.get(isinstance(arg, Message) and Dialog.extractId(arg) or str(arg), None)
```

The `findTransaction` method can be used to locate an existing `Transaction` object given the transaction identifier string. The method returns `None,` if a transaction is not found.

```
    def findTransaction(self, id):
        return self.transactions.get(id, None)
```

The `findOtherTransaction` method returns another transaction other than the specified original transaction `orig` that matches the given request `r` of type `Message`. Although the implementation described below iterates through all transactions to find a match, a more efficient hash table can be built for such operation. The `Transaction.equals` method is invoked to compare the request against a transaction such that it is different from the original transaction. The method returns `None` if no other transaction is found. The method is useful in request merging and loop-detection logic in the UAS implementation.

```
    def findOtherTransaction(self, r, orig):
        for t in self.transactions.values():
            if t != orig and Transaction.equals(t, r, orig): return t
        return None
```

Wrapper for application callbacks

To finish up the implementation of the `Stack` class, we define a bunch of wrapper methods to shield the callback invocation between the rest of the SIP implementation and the application. The rest of the SIP layers invoke these wrapper methods on the `Stack`, which in turn invokes the application callback. These wrapper methods are typically invoked by UAS/UAC or dialog layer. This allows us to be consistent across these

callbacks by supplying the `Stack` reference as the last parameter in all the application callbacks, as we had discussed earlier.

```
    def createServer(self, request, uri):   return self.app.createServer(request, uri, self)
    def sending(self, ua, message):             self.app.sending(ua, message, self)
    def receivedRequest(self, ua, request):     self.app.receivedRequest(ua, request, self)
    def receivedResponse(self, ua, response): self.app.receivedResponse(ua, response, self)
    def cancelled(self, ua, request):           self.app.cancelled(ua, request, self)
    def dialogCreated(self, dialog, ua):        self.app.dialogCreated(dialog, ua, self)
    def authenticate(self, ua, header):     return self.app.authenticate(ua, header, self)
    def createTimer(self, obj):             return self.app.createTimer(obj, self)
```

The `Stack` layer we described controls the main core logic of the SIP implementation as well as the interface between the SIP implementation and the application. The other layers such as transaction, UAC/UAS and dialog are very precisely described in RFC3261, hence should be easier to implement compared to the `Stack` class. We describe the implementation of the other layers next.

User agent (UAC and UAS)

> From RFC3261 p.34 – A user agent represents an end system. It contains a user agent client (UAC), which generates requests, and a user agent server (UAS), which responds to them. A UAC is capable of generating a request based on some external stimulus (the user clicking a button, or a signal on a PSTN line) and processing a response. A UAS is capable of receiving a request and generating a response based on user input, external stimulus, the result of a program execution, or some other mechanism.

We implement UAC and UAS using a single class `UserAgent`. The object behaves differently depending on whether it is a client (UAC) or a server (UAS). The property `server` identifies whether it is a server (`True`) or a client (`False`). A UAC or a UAS can create a dialog on certain conditions, such as when a 2xx-class response to an INVITE UAC is received or when a 2xx class response to an INVITE UAS is sent.

> UAC and UAS procedures depend strongly on two factors. First, based on whether the request or response is inside or outside of a dialog, and second, based on the method of a request.

The procedure to send a request or response or process an incoming request or response in a user agent is slightly different than that in a dialog context. But there are a number of properties that are common between a user agent and a dialog. Instead of defining separate independent classes for implementing a user agent and a dialog, we derive the `Dialog` class from the `UserAgent` class. Most of the properties defined here are reused in the derived class `Dialog`.

Constructor and properties

The constructor for a UAS takes the original incoming request `Message` whereas for a UAC it should not. The first argument is a reference to the associated `Stack` object so that various functions on the stack can be performed. The second argument is the optional original request needed for constructing a UAS. The last argument in the constructor defines whether this is a UAS or UAC.

```
class UserAgent(object):
    def __init__(self, stack, request=None, server=None):
        self.stack, self.request = stack, request
```

```
        self.server = server if server != None else (request != None)
```

Each user agent stores a reference to the last `Transaction` associated with this user agent. It also stores a reference to the cancel request `Message` is it was sent or needs to be sent. We will describe these properties later when they are used.

```
        self.transaction, self.cancelRequest = None, None
```

The `callId` property refers to the unique `Call-ID` for this user agent or dialog. It is either extracted from the existing request `Message` for a UAS, or created randomly on the stack context for a UAC.

```
        self.callId = request['Call-ID'].value if request and request['Call-ID'] else stack.newCallId
```

Each user agent or dialog has `localParty` and `remoteParty` properties that refer to the SIP `Address` of the local entity or the remote entity. These addresses are put in the `From` and `To` headers of the generated request or extracted from the To and From headers of the received request, respectively.

```
        self.remoteParty = request.From.value if request and request.From else None
        self.localParty = request.To.value if request and request.To else None
```

The `To` and `From` headers also need a `tag` parameter. The user agent and dialog objects store the unique `localTag` and `remoteTag` properties. The local `tag` is derived from the unique tag associated with the stack context, but with additional randomness to it. This allows the `tag` parameter to uniquely identify the stack but also be different for different dialogs or user agents.

```
        self.localTag, self.remoteTag  = stack.tag + str(random.randint(0,10*10)), None
```

The `subject` property is used for the `Subject` header in the SIP request within this user agent or dialog.

```
        self.subject = request.Subject.value if request and request.Subject else None
```

> From RFC3261 – A dialog contains certain pieces of state needed for further message transmissions within the dialog. This state consists of the dialog ID, a local sequence number (used to order requests from the UA to its peer), a remote sequence number (used to order requests from its peer to the UA), a local URI, a remote URI, remote target, a boolean flag called "secure", and a route set, which is an ordered list of URIs. The route set is the list of servers that need to be traversed to send a request to the peer.

The `secure` property indicates whether this user agent or dialog is operating on a secure connection using the “sips:” URI scheme or not.

```
        self.secure = (request and request.uri.scheme == 'sips')
```

The outgoing SIP requests should have a `Max-Forwards` header to limit the number of SIP hops to traverse among intermediate proxies. Similarly, the request can have a `Route` header to pre-determine the

SIP hops to traverse for a request. These headers are generated using the `maxForwards` and `routeSet` properties. The default value of `Max-Forwards` header is 70. The `routeSet` is either derived using the `Record-Route` header as described later or pre-set by the application, e.g., for setting the outbound proxy.

```
self.maxForwards, self.routeSet = 70, []
```

The exact host-port to be used for sending a request or response is derived from the DNS lookup for outgoing requests, and for response from various header fields of incoming requests. The `remoteCandidates` property stores the list of potential DNS entries to try for sending an initial request. The `localTarget` and the `remoteTarget` properties store the local and remote addresses to which a request or response will be sent in a user agent or dialog.

```
self.localTarget, self.remoteTarget, self.remoteCandidates = None, None, None
```

The local sequence number is incremented for subsequent requests in a dialog. The remote sequence number is used to detect whether an incoming request in a dialog is obsolete and should be ignored or not. These pieces of state are stored in the `localSeq` and `remoteSeq` properties, respectively.

```
self.localSeq, self.remoteSeq = 0, 0
```

Certain outgoing requests or responses need to have a `Contact` header that represents the local user's contact, so that incoming request that be sent on that contact for subsequent requests in this dialog. For example, the UAS can return a `Contact` header in the 2xx-class response to an incoming INVITE request. The remote party should then use the address specified in the `Contact` header to send future requests such as BYE within this dialog, provided the constraints of route set allows it. We define the `contact` property to store this local SIP address, such that the `user` part of the address is derived from the `localParty` address whereas the `host` and `port` parts are derived from the local listening point in the stack context.

```
self.contact = Address(str(stack.uri))
if self.localParty and self.localParty.uri.user: self.contact.uri.user = self.localParty.uri.user
```

For example, if the local party is "sip:kundan@example.net" and the stack's listening address is "sip:192.1.2.3:5080" then the `contact` property represents the address "sip:kundan@192.1.2.3:5080".

Besides the above properties we also need two additional properties specific for our implementation. In particular the `autoack` property indicates whether the implementation should automatically send an ACK to the 2xx-class response to an incoming INVITE request, or whether the implementation should let the application send the ACK explicitly. If the implementation sends the ACK automatically, then it performs some functions of the application as per RFC 3261, because the specification defines that ACK for 2xx-class response to INVITE should be sent end-to-end by the application. However, in practice the application may not want to deal with the specifics of SIP implementation. By default we let the SIP implementation automatically send the ACK, but if the application does want to be in control of sending the ACK, e.g., to change the message body in the ACK, then it can set the `autoack` property to `False`.

```
self.autoack = True # whether to send an ACK to 200 OK of INVITE automatically or let application send it.
```

Finally, the `auth` property stores the various authentication contexts such as user credentials and other properties. The authentication context is used for authenticating an outgoing request that has been challenged by the remote party.

```
    self.auth = dict() # to store authentication context
```

A string representation of the object just displays the `Call-ID` property of the object and the name of the class, whether it is a `Dialog` or a `UserAgent`.

```
def __repr__(self):
    return '<%s call-id=%s>'%(isinstance(self, Dialog) and 'Dialog' or 'UserAgent', self.callId)
```

Generating the request

The application can create a new out-of-dialog request using the `createRequest` method on a newly created `UserAgent` object. The method sets the `UserAgent` object as a UAC.

> From RFC3261 p.35 – Examples of requests sent outside of a dialog include an INVITE to establish a session and an OPTIONS to query for capabilities.

```
ua = UserAgent(stack)
…
m = ua.createRequest("INVITE", sdp, "application/sdp")
```

The method first checks whether the needed properties such as `remoteParty` and `localParty` are set correctly or not. It is an error if the remote party address is unknown when creating a request. If the local party address is unknown, the implementation uses the anonymous address.

> The From header field allows for a display name. A UAC SHOULD use the display name "Anonymous", along with a syntactically correct, but otherwise meaningless URI (like sip:thisis@anonymous.invalid), if the identity of the client is to remain hidden.

```
def createRequest(self, method, content=None, contentType=None):
    self.server = False
    if not self.remoteParty: raise ValueError, 'No remoteParty for UAC'
    if not self.localParty: self.localParty = Address('"Anonymous" <sip:anonymous@anonymous.invalid>')
```

> (UAC) The initial Request-URI of the message SHOULD be set to the value of the URI in the To field.
>
> One notable exception is the REGISTER method; behavior for setting the Request-URI of REGISTER is given in Section 10. (In Section 10) The "userinfo" and "@" components of the SIP URI MUST NOT be present.

```
    uri = URI(str(self.remoteTarget if self.remoteTarget else self.remoteParty.uri)) # TODO: use original URI for ACK
    if method == 'REGISTER': uri.user = None # no uri.user in REGISTER
```

```
if not self.secure and uri.secure: self.secure = True
if method != 'ACK' and method != 'CANCEL': self.localSeq = self.localSeq + 1
```

The To header field first and foremost specifies the desired "logical" recipient of the request, or the address-of-record of the user or resource that is the target of this request. This may or may not be the ultimate recipient of the request.

(UAC) A request outside of a dialog MUST NOT contain a To tag; the tag in the To field of a request identifies the peer of the dialog. Since no dialog is established, no tag is present.

(Dialog) The URI in the To field of the request MUST be set to the remote URI from the dialog state.

```
To = Header(str(self.remoteParty), 'To')
To.value.uri.secure = self.secure
```

The From header field indicates the logical identity of the initiator of the request, possibly the user's address-of-record. Like the To header field, it contains a URI and optionally a display name. It is used by SIP elements to determine which processing rules to apply to a request (for example, automatic call rejection). As such, it is very important that the From URI not contain IP addresses or the FQDN of the host on which the UA is running, since these are not logical names.

(UAC) The From field MUST contain a new "tag" parameter, chosen by the UAC.

(Dialog) The From URI of the request MUST be set to the local URI from the dialog state. The tag in the From header field of the request MUST be set to the local tag of the dialog ID. If the value of the remote or local tags is null, the tag parameter MUST be omitted from the To or From header fields, respectively.

```
From = Header(str(self.localParty), 'From')
From.value.uri.secure = self.secure
From.tag = self.localTag
```

The CSeq header field serves as a way to identify and order transactions. It consists of a sequence number and a method. The method MUST match that of the request. For non-REGISTER requests outside of a dialog, the sequence number value is arbitrary. The sequence number value MUST be expressible as a 32-bit unsigned integer and MUST be less than 2**31. As long as it follows the above guidelines, a client may use any mechanism it would like to select CSeq header field values.

```
CSeq = Header(str(self.localSeq) + ' ' + method, 'CSeq')
```

The Call-ID header field acts as a unique identifier to group together a series of messages. It MUST be the same for all requests and responses sent by either UA in a dialog.

(Dialog) The Call-ID of the request MUST be set to the Call-ID of the dialog.

```
CallId = Header(self.callId, 'Call-ID')
```

The Max-Forwards header field serves to limit the number of hops a request can transit on the way to its destination. It consists of an integer that is decremented by one at each hop. If the Max-Forwards value reaches 0 before the request reaches its destination, it will be rejected with a 483(Too Many Hops) error response.

A UAC MUST insert a Max-Forwards header field into each request it originates with a value that SHOULD be 70. This number was chosen to be sufficiently large to guarantee that a request would not be dropped in any SIP network when there were no loops, but not so large as to consume proxy resources when a loop does occur. Lower values should be used with caution and only in networks where topologies are known by the UA.

```
MaxForwards = Header(str(self.maxForwards), 'Max-Forwards')
```

When the UAC creates a request, it MUST insert a Via into that request. The protocol name and protocol version in the header field MUST be SIP and 2.0, respectively. The Via header field value MUST contain a branch parameter. This parameter is used to identify the transaction created by that request. This parameter is used by both the client and the server.

```
Via = self.stack.createVia(self.secure)
Via.branch = Transaction.createBranch([To.value, From.value, CallId.value, CSeq.number], False)
# Transport adds other parameters such as maddr, ttl

if not self.localTarget:
   self.localTarget = self.stack.uri.dup()
   self.localTarget.user = self.localParty.uri.user
```

(UAC) The Contact header field provides a SIP or SIPS URI that can be used to contact that specific instance of the UA for subsequent requests. The scope of the Contact is global. That is, the Contact header field value contains the URI at which the UA would like to receive requests, and this URI MUST be valid even if used in subsequent requests outside of any dialogs. If the Request-URI or top Route header field value contains a SIPS URI, the Contact header field MUST contain a SIPS URI as well.

(Dialog) A UAC SHOULD include a Contact header field in any target refresh requests within a dialog, and unless there is a need to change it, the URI SHOULD be the same as used in previous requests within the dialog. If the "secure" flag is true, that URI MUST be a SIPS URI.

```
Contact = Header(str(self.localTarget), 'Contact')
Contact.value.uri.secure = self.secure
```

A valid SIP request formulated by a UAC MUST, at a minimum, contain the following header fields: To, From, CSeq, Call-ID, Max-Forwards, and Via; all of these header fields are mandatory in all SIP requests.

```
headers = [To, From, CSeq, CallId, MaxForwards, Via, Contact]
```

In some special circumstances, the presence of a pre-existing route set can affect the Request-URI of the message. A pre-existing route set is an ordered set of URIs that identify a chain of servers, to which a UAC will send outgoing requests that are outside of a dialog. Commonly, they are configured on the UA by a user or service provider manually, or through some other non-SIP mechanism. When a provider wishes to configure a UA with an outbound proxy, it is RECOMMENDED that this be done by providing it with a pre-existing route set with a single URI, that of the outbound proxy.

When a pre-existing route set is present, the procedures for populating the Request-URI and Route header field detailed in Section 12.2.1.1 MUST be followed (even though there is no dialog), using the desired Request-URI as the remote target URI.

```
    if self.routeSet:
        for route in map(lambda x: Header(str(x), 'Route'), self.routeSet):
            route.value.uri.secure = self.secure
            headers.append(route)
```

If the UAC supports extensions to SIP that can be applied by the server to the response, the UAC SHOULD include a Supported header field in the request listing the option tags (Section 19.2) for those extensions.

If the UAC wishes to insist that a UAS understand an extension that the UAC will apply to the request in order to process the request, it MUST insert a Require header field into the request listing the option tag for that extension. If the UAC wishes to apply an extension to the request and insist that any proxies that are traversed understand that extension, it MUST insert a Proxy-Require header field into the request listing the option tag for that extension.

```
    # app adds other headers such as Supported, Require and Proxy-Require
```

SIP requests MAY contain a MIME-encoded message-body. Regardless of the type of body that a request contains, certain header fields must be formulated to characterize the contents of the body.

```
    if contentType:
        headers.append(Header(contentType, 'Content-Type'))
    self.request = Message.createRequest(method, str(uri), headers, content)
    return self.request
```

Generating a REGISTER request

To: The To header field contains the address of record whose registration is to be created, queried, or modified. The To header field and the Request-URI field typically differ, as the former contains a user name. This address-of-record MUST be a SIP URI or SIPS URI.

From: The From header field contains the address-of-record of the person responsible for the registration. The value is the same as the To header field unless the request is a third-party registration.

```
def createRegister(self, aor):
    if aor: self.remoteParty = Address(str(aor))
    if not self.localParty: self.localParty = Address(str(self.remoteParty))
```

Except as noted, the construction of the REGISTER request and the behavior of clients sending a REGISTER request is identical to the general UAC behavior

```
    return self.createRequest('REGISTER')
```

Sending the request

From RFC3261 p.41 – The destination for the request is then computed. Unless there is local policy specifying otherwise, the destination MUST be determined by applying the DNS procedures described in [4] as follows. If

the first element in the route set indicated a strict router (resulting in forming the request as described in Section 12.2.1.1), the procedures MUST be applied to the Request-URI of the request. Otherwise, the procedures are applied to the first Route header field value in the request (if one exists), or to the request's Request-URI if there is no Route header field present. These procedures yield an ordered set of address, port, and transports to attempt. Independent of which URI is used as input to the procedures of [4], if the Request-URI specifies a SIPS resource, the UAC MUST follow the procedures of [4] as if the input URI were a SIPS URI.

Local policy MAY specify an alternate set of destinations to attempt. If the Request-URI contains a SIPS URI, any alternate destinations MUST be contacted with TLS. Beyond that, there are no restrictions on the alternate destinations if the request contains no Route header field. This provides a simple alternative to a pre-existing route set as a way to specify an outbound proxy. However, that approach for configuring an outbound proxy is NOT RECOMMENDED; a pre-existing route set with a single URI SHOULD be used instead. If the request contains a Route header field, the request SHOULD be sent to the locations derived from its topmost value, but MAY be sent to any server that the UA is certain will honor the Route and Request-URI policies specified in this document (as opposed to those in RFC 2543). In particular, a UAC configured with an outbound proxy SHOULD attempt to send the request to the location indicated in the first Route header field value instead of adopting the policy of sending all messages to the outbound proxy.

```
def sendRequest(self, request):
   if not self.request and request.method == 'REGISTER':
      if not self.transaction and self.transaction.state != 'completed' and self.transaction.state != 'terminated':
         raise ValueError, 'Cannot re-REGISTER since pending registration'
   self.request = request

   if not request.Route: self.remoteTarget = request.uri
   target = self.remoteTarget

   if request.Route:
      routes = request.all('Route')
      if len(routes) > 0:
         target = routes[0].value.uri
         if not target or 'lr' not in target.param: # strict route
            if _debug: print 'strict route target=', target, 'routes=', routes
            del routes[0] # ignore first route
            if len(routes) > 0:
               if _debug: print 'appending our route'
               routes.append(Header(str(request.uri), 'Route'))
            request.Route = routes
            request.uri = target;

   # TODO: remove any Route header in REGISTER request

   self.stack.sending(self, request)
```

The UAC SHOULD follow the procedures defined in [4] for stateful elements, trying each address until a server is contacted. Each try constitutes a new transaction, and therefore each carries a different topmost Via header field value with a new branch parameter. Furthermore, the transport value in the Via header field is set to whatever transport was determined for the target server.

```
   # TODO: replace the following with RFC3263 to return multiple candidates. Add TCP and UDP and if possible TLS.
   dest = target.dup()
   dest.port = target.port or target.secure and 5061 or 5060
   if not isIPv4(dest.host):
```

```
            try: dest.host = gethostbyname(dest.host)
            except: pass
        if isIPv4(dest.host):
            self.remoteCandidates = [dest]

        # continue processing as if we received multiple candidates
        if not self.remoteCandidates or len(self.remoteCandidates) == 0:
            self.error(None, 'cannot resolve DNS target')
            return
        target = self.remoteCandidates.pop(0)
        if self.request.method != 'ACK':   # start a client transaction to send the request
            self.transaction = Transaction.createClient(self.stack, self, self.request, self.stack.transport, target.hostPort)
        else: # directly send ACK on transport layer
            self.stack.send(self.request, target.hostPort)
```

From RFC3261 p.42 – In some cases, the response returned by the transaction layer will not be a SIP message, but rather a transaction layer error. When a timeout error is received from the transaction layer, it MUST be treated as if a 408 (Request Timeout) status code has been received. If a fatal transport error is reported by the transport layer (generally, due to fatal ICMP errors in UDP or connection failures in TCP), the condition MUST be treated as a 503 (Service Unavailable) status code.

When the transaction times out we try the next candidate address.

```
    def timeout(self, transaction):
        if transaction and transaction != self.transaction: # invalid transaction
            return
        self.transaction = None
        if not self.server: # UAC
            if self.remoteCandidates and len(self.remoteCandidates)>0:
                self.retryNextCandidate()
            else:
                self.receivedResponse(None, Message.createResponse(408, 'Request timeout', None, None, self.request))
```

The following method is invoked to try the next candidate address from the DNS result for the destination.

```
    def retryNextCandidate(self):
        if not self.remoteCandidates or len(self.remoteCandidates) == 0:
            raise ValueError, 'No more DNS resolved address to try'
        target = URI(self.remoteCandiates.pop(0))
        self.request.first('Via').branch += 'A' # so that we create a different new transaction
        transaction = Transaction.createClient(self.stack, self, self.request, self.stack.transport, target.hostPort)
```

A transport error in sending a request is treated as “503 Service unavailable”.

```
    def error(self, transaction, error):
        if transaction and transaction != self.transaction: # invalid transaction
            return
        self.transaction = None
        if not self.server: # UAC
            if self.remoteCandidates and len(self.remoteCandidates)>0:
                self.retryNextCandidate()
```

```
        else:
            self.receivedResponse(None, Message.createResponse(503, 'Service unavailable - ' + error, None, None,
self.request))
```

Processing responses

From RFC3261 p.42 – Responses are first processed by the transport layer and then passed up to the transaction layer. The transaction layer performs its processing and then passes the response up to the TU. The majority of response processing in the TU is method specific. However, there are some general behaviors independent of the method.

```
def receivedResponse(self, transaction, response):
    if transaction and transaction != self.transaction:
        if _debug: print 'Invalid transaction received %r!=%r'%(transaction, self.transaction)
        return
```

If more than one Via header field value is present in a response, the UAC SHOULD discard the message.

```
    if len(response.all('Via')) > 1:
        raise ValueError, 'More than one Via header in response'
    if response.is1xx:
        if self.cancelRequest:
            cancel = Transaction.createClient(self.stack, self, self.cancelRequest, transaction.transport, transaction.remote)
            self.cancelRequest = None
        else:
            self.stack.receivedResponse(self, response)
```

If a 401 (Unauthorized) or 407 (Proxy Authentication Required) response is received, the UAC SHOULD follow the authorization procedures of Section 22.2 and Section 22.3 to retry the request with credentials.

```
    elif response.response == 401 or response.response == 407: # authentication challenge
        if not self.authenticate(response, self.transaction): # couldn't authenticate
            self.stack.receivedResponse(self, response)
    else:
        if self.canCreateDialog(self.request, response):
            dialog = Dialog.createClient(self.stack, self.request, response, transaction)
            self.stack.dialogCreated(dialog, self)
            self.stack.receivedResponse(dialog, response)
            if self.autoack and self.request.method == 'INVITE':
                dialog.sendRequest(dialog.createRequest('ACK'))
        else:
            self.stack.receivedResponse(self, response)
```

The global method `canCreateDialog` determines whether a dialog can be created out of the given response for the given original request. The current implementation creates a dialog for a 2xx-class response for the original INVITE or SUBSCRIBE requests.

Dialogs are created through the generation of non-failure responses to requests with specific methods. Within this specification, only 2xx and 101-199 responses with a To tag, where the request was INVITE, will establish a dialog. A dialog established by a non-final response to a request is in the "early" state and it is called an early dialog.

In our implementation we do not support early dialogs.

```
@staticmethod
def canCreateDialog(request, response):
    return response.is2xx and (request.method == 'INVITE' or request.method == 'SUBSCRIBE')
```

Processing requests

From RFC3261 p46 – When a request outside of a dialog is processed by a UAS, there is a set of processing rules that are followed, independent of the method.

Note that request processing is atomic. If a request is accepted, all state changes associated with it MUST be performed. If it is rejected, all state changes MUST NOT be performed.

UASs SHOULD process the requests in the order of the steps that follow in this section (that is, starting with authentication, then inspecting the method, the header fields, and so on throughout the remainder of this section).

```
def receivedRequest(self, transaction, request):
    if transaction and self.transaction and transaction != self.transaction and request.method != 'CANCEL':
        raise ValueError, 'Invalid transaction for received request'
    self.server = True # this becomes a UAS
```

Once a request is authenticated (or authentication is skipped), the UAS MUST inspect the method of the request. If the UAS recognizes but does not support the method of a request, it MUST generate a 405 (Method Not Allowed) response. Procedures for generating responses are described in Section 8.2.6. The UAS MUST also add an Allow header field to the 405 (Method Not Allowed) response. The Allow header field MUST list the set of methods supported by the UAS generating the message. The Allow header field is presented in Section 20.5.

If the method is one supported by the server, processing continues.

```
#if request.method == 'REGISTER':
#    response = transaction.createResponse(405, 'Method not allowed')
#    response.Allow = Header('INVITE, ACK, CANCEL, BYE', 'Allow') # TODO make this configurable
#    transaction.sendResponse(response)
#    return
```

However, the Request-URI identifies the UAS that is to process the request. If the Request-URI uses a scheme not supported by the UAS, it SHOULD reject the request with a 416 (Unsupported URI Scheme) response.

```
if request.uri.scheme not in ['sip', 'sips']:
    transaction.sendResponse(transaction.createResponse(416, 'Unsupported URI scheme'))
    return
```

If the request has no tag in the To header field, the UAS core MUST check the request against ongoing transactions. If the From tag, Call-ID, and CSeq exactly match those associated with an ongoing transaction, but the request does not match that transaction, the UAS core SHOULD generate a 482 (Loop Detected) response and pass it to the server transaction.

```
if 'tag' not in request.To: # out of dialog request
    if self.stack.findOtherTransaction(request, transaction): # request merging?
        transaction.sendResponse(transaction.createResponse(482, "Loop detected - found another transaction"))
        return
```

Assuming the UAS decides that it is the proper element to process the request, it examines the Require header field, if present.

The Require header field is used by a UAC to tell a UAS about SIP extensions that the UAC expects the UAS to support in order to process the request properly. If a UAS does not understand an option-tag listed in a Require header field, it MUST respond by generating a response with status code 420 (Bad Extension). The UAS MUST add an Unsupported header field, and list in it those options it does not understand amongst those in the Require header field of the request. Note that Require and Proxy-Require MUST NOT be used in a SIP CANCEL request, or in an ACK request sent for a non-2xx response. These header fields MUST be ignored if they are present in these requests.

An ACK request for a 2xx response MUST contain only those Require and Proxy-Require values that were present in the initial request.

```
if request.Require: # TODO let the application handle Require header
    if request.method != 'CANCEL' and request.method != 'ACK':
        response = transaction.createResponse(420, 'Bad extension')
        response.Unsupported = Header(str(request.Require.value), 'Unsupported')
        transaction.sendResponse(response)
        return
if transaction: self.transaction = transaction # store it
```

The CANCEL method requests that the TU at the server side cancel a pending transaction. The TU determines the transaction to be cancelled by taking the CANCEL request, and then assuming that the request method is anything but CANCEL or ACK and applying the transaction matching procedures of Section 17.2.3. The matching transaction is the one to be cancelled.

The processing of a CANCEL request at a server depends on the type of server. A stateless proxy will forward it, a stateful proxy might respond to it and generate some CANCEL requests of its own, and a UAS will respond to it. See Section 16.10 for proxy treatment of CANCEL.

A UAS first processes the CANCEL request according to the general UAS processing described in Section 8.2. However, since CANCEL requests are hop-by-hop and cannot be resubmitted, they cannot be challenged by the server in order to get proper credentials in an Authorization header field. Note also that CANCEL requests do not contain a Require header field.

If the UAS did not find a matching transaction for the CANCEL according to the procedure above, it SHOULD respond to the CANCEL with a 481 (Call Leg/Transaction Does Not Exist). If the transaction for the original request still exists, the behavior of the UAS on receiving a CANCEL request depends on whether it has already sent a final response for the original request. If it has, the CANCEL request has no effect on the processing of the original request, no effect on any session state, and no effect on the responses generated for the original request. If the UAS has not issued a final response for the original request, its behavior depends on the method of the original request. If the original request was an INVITE, the UAS SHOULD immediately respond to the

INVITE with a 487 (Request Terminated). A CANCEL request has no impact on the processing of transactions with any other method defined in this specification.

Regardless of the method of the original request, as long as the CANCEL matched an existing transaction, the UAS answers the CANCEL request itself with a 200 (OK) response. This response is constructed following the procedures described in Section 8.2.6 noting that the To tag of the response to the CANCEL and the To tag in the response to the original request SHOULD be the same. The response to CANCEL is passed to the server transaction for transmission.

```
if request.method == 'CANCEL':
    original = self.stack.findTransaction(Transaction.createId(transaction.branch, 'INVITE'))
    if not original:
        transaction.sendResponse(transaction.createResponse)
        return
    if original.state == 'proceeding' or original.state == 'trying':
        original.sendResponse(original.createResponse(487, 'Request terminated'))
    transaction.sendResponse(transaction.createResponse(200, 'OK')) # CANCEL response
    # TODO: the To tag must be same in the two responses
```

Finally, the request is delivered to the application for further processing after the UAS procedures are applied.

```
self.stack.receivedRequest(self, request)
```

Generating the response

From RFC3261 p.49 – When a UAS wishes to construct a response to a request, it follows the general procedures detailed in the following subsections. Additional behaviors specific to the response code in question, which are not detailed in this section, may also be required.

Once all procedures associated with the creation of a response have been completed, the UAS hands the response back to the server transaction from which it received the request.

```
def sendResponse(self, response, responsetext=None, content=None, contentType=None, createDialog=True):
    if not self.request:
        raise ValueError, 'Invalid request in sending a response'
    if isinstance(response, int):
        response = self.createResponse(response, responsetext, content, contentType)
```

When a UAS responds to a request with a response that establishes a dialog (such as a 2xx to INVITE), the UAS MUST copy all Record-Route header field values from the request into the response (including the URIs, URI parameters, and any Record-Route header field parameters, whether they are known or unknown to the UAS) and MUST maintain the order of those values.

```
    if createDialog and self.canCreateDialog(self.request, response):
        if self.request['Record-Route']: response['Record-Route'] = self.request['Record-Route']
```

The UAS MUST add a Contact header field to the response. The Contact header field contains an address where the UAS would like to be contacted for subsequent requests in the dialog (which includes the ACK for a 2xx response in the case of an INVITE). Generally, the host portion of this URI is the IP address or FQDN of the host. The URI provided in the Contact header field MUST be a SIP or SIPS URI. If the request that initiated the dialog contained a SIPS URI in the Request-URI or in the top Record-Route header field value, if there was any, or the Contact header field if there was no Record-Route header field, the Contact header field in the response MUST be a SIPS URI. The URI SHOULD have global scope (that is, the same URI can be used in messages outside this dialog). The same way, the scope of the URI in the Contact header field of the INVITE is not limited to this dialog either. It can therefore be used in messages to the UAC even outside this dialog.

```
    if not response.Contact:
        contact = Address(str(self.contact))
        if not contact.uri.user: contact.uri.user = self.request.To.value.uri.user
        contact.uri.secure = self.secure
        response.Contact = Header(str(contact), 'Contact')
```

The UAS then constructs the state of the dialog. This state MUST be maintained for the duration of the dialog.

```
        dialog = Dialog.createServer(self.stack, self.request, response, self.transaction)
        self.stack.dialogCreated(dialog, self)
        self.stack.sending(dialog, response)
    else:
        self.stack.sending(self, response)

    if not self.transaction: # send on transport
        self.stack.send(response, response.first('Via').viaUri.hostPort)
    else:
        self.transaction.sendResponse(response)
```

Additionally, the UAS MUST add a tag to the To header field in the response (with the exception of the 100 (Trying) response, in which a tag MAY be present). This serves to identify the UAS that is responding, possibly resulting in a component of a dialog ID. The same tag MUST be used for all responses to that request, both final and provisional (again excepting the 100 (Trying)).

```
def createResponse(self, response, responsetext, content=None, contentType=None):
    if not self.request:
        raise ValueError, 'Invalid request in creating a response'
    response = Message.createResponse(response, responsetext, None, content, self.request)
    if contentType: response['Content-Type'] = Header(contentType, 'Content-Type')
    if response.response != 100 and 'tag' not in response.To: response.To['tag'] = self.localTag
    return response;
```

Cancelling a request

From RFC3261 p.53 – The CANCEL request, as the name implies, is used to cancel a previous request sent by a client. Specifically, it asks the UAS to cease processing the request and to generate an error response to that request. CANCEL has no effect on a request to which a UAS has already given a final response. Because of this, it is most useful to CANCEL requests to which it can take a server long time to respond. For this reason,

CANCEL is best for INVITE requests, which can take a long time to generate a response. In that usage, a UAS that receives a CANCEL request for an INVITE, but has not yet sent a final response, would "stop ringing", and then respond to the INVITE with a specific error response (a 487).

The following procedures are used to construct a CANCEL request. The Request-URI, Call-ID, To, the numeric part of CSeq, and From header fields in the CANCEL request MUST be identical to those in the request being cancelled, including tags. A CANCEL constructed by a client MUST have only a single Via header field value matching the top Via value in the request being cancelled. Using the same values for these header fields allows the CANCEL to be matched with the request it cancels (Section 9.2 indicates how such matching occurs). However, the method part of the CSeq header field MUST have a value of CANCEL. This allows it to be identified and processed as a transaction in its own right (See Section 17).

```
def sendCancel(self):
    if not self.transaction:
        raise ValueError, 'No transaction for sending CANCEL'
    self.cancelRequest = self.transaction.createCancel()
```

If the request being cancelled contains a Route header field, the CANCEL request MUST include that Route header field's values. The CANCEL request MUST NOT contain any Require or Proxy-Require header fields.

Once the CANCEL is constructed, the client SHOULD check whether it has received any response (provisional or final) for the request being cancelled (herein referred to as the "original request").

If no provisional response has been received, the CANCEL request MUST NOT be sent; rather, the client MUST wait for the arrival of a provisional response before sending the request. If the original request has generated a final response, the CANCEL SHOULD NOT be sent, as it is an effective no-op, since CANCEL has no effect on requests that have already generated a final response. When the client decides to send the CANCEL, it creates a client transaction for the CANCEL and passes it the CANCEL request along with the destination address, port, and transport. The destination address, port, and transport for the CANCEL MUST be identical to those used to send the original request.

```
    if self.transaction.state != 'trying' and self.transaction.state != 'calling':
        if self.transaction.state == 'proceeding':
            transaction = Transaction.createClient(self.stack, self, self.cancelRequest, self.transaction.transport,
self.transaction.remote)
        self.cancelRequest = None
    # else don't send until 1xx is received
```

Authentication

From RFC3261 p.196 – When the originating UAC receives the 401 (Unauthorized), it SHOULD, if it is able, re-originate the request with the proper credentials. The UAC may require input from the originating user before proceeding. Once authentication credentials have been supplied (either directly by the user, or discovered in an internal keyring), UAs SHOULD cache the credentials for a given value of the To header field and "realm" and attempt to re-use these values on the next request for that destination. UAs MAY cache credentials in any way they would like.

When a request receives a 401 or 407 response in a UAC, we invoke the `authenticate` method. If the application has supplied the local user's credentials, then we use that to resend the request in a new transaction, in the same UAC. If the request was resent, then it returns `True`, otherwise it returns `False`.

```
def authenticate(self, response, transaction):
```

```
    a = response.first('WWW-Authenticate') or response.first('Proxy-Authenticate') or None
    if not a:
      return False
    request = Message(str(transaction.request)) # construct a new message

    resend, present = False, False
    for b in request.all('Authorization', 'Proxy-Authorization'):
      if a.realm == b.realm and (a.name == 'WWW-Authenticate' and b.name == 'Authorization' or a.name == 'Proxy-
Authenticate' and b.name == 'Proxy-Authorization'):
        present = True
        break

    if not present and 'realm' in a: # prompt for password
      result = self.stack.authenticate(self, a)
      if not result or 'password' not in a and 'hashValue' not in a:
        return False
```

Once credentials have been located, any UA that wishes to authenticate itself with a UAS or registrar -- usually, but not necessarily, after receiving a 401 (Unauthorized) response -- MAY do so by including an Authorization header field with the request. The Authorization field value consists of credentials containing the authentication information of the UA for the realm of the resource being requested as well as parameters required in support of authentication and replay protection.

```
      value = createAuthorization(a.value, a.username, a.password, str(request.uri), self.request.method,
self.request.body, self.auth)
      if value:
        request.insert(Header(value, (a.name == 'WWW-Authenticate') and 'Authorization' or 'Proxy-Authorization'),
True)
        resend = True
```

When a UAC resubmits a request with its credentials after receiving a 401 (Unauthorized) or 407 (Proxy Authentication Required) response, it MUST increment the CSeq header field value as it would normally when sending an updated request.

```
    if resend:
      self.localSeq = self.localSeq + 1
      request.CSeq = Header(str(self.localSeq) + ' ' + request.method, 'CSeq')
      request.first('Via').branch = Transaction.createBranch(request, False)
      self.request = request
      self.transaction = Transaction.createClient(self.stack, self, self.request, self.transaction.transport,
self.transaction.remote)
      return True
    else:
      return False;
```

Dialog

From RFC3261 p.69 – A key concept for a user agent is that of a dialog. A dialog represents a peer-to-peer SIP relationship between two user agents that persists for some time. The dialog facilitates sequencing of messages between the user agents and proper routing of requests between both of them. The dialog represents a context in which to interpret SIP messages.

Since a number of properties are shared between the UAC/UAS and the dialog context, we derive the `Dialog` class from the `UserAgent` class.

```
class Dialog(UserAgent):
  @staticmethod
  def createServer(stack, request, response, transaction):
    d = Dialog(stack, request, True)
    d.request = request
```

The route set MUST be set to the list of URIs in the Record-Route header field from the request, taken in order and preserving all URI parameters. If no Record-Route header field is present in the request, the route set MUST be set to the empty set. This route set, even if empty, overrides any pre-existing route set for future requests in this dialog. The remote target MUST be set to the URI from the Contact header field of the request.

```
    d.routeSet = request.all('Record-Route') if request['Record-Route'] else None
    while d.routeSet and isMulticast(d.routeSet[0].value.uri.host): # remove any multicast address from top of the list.
      if _debug: print 'deleting top multicast routeSet', d.routeSet[0]
      del d.routeSet[0]
      if len(d.routeSet) == 0: d.routeSet = None
```

If the request arrived over TLS, and the Request-URI contained a SIPS URI, the "secure" flag is set to TRUE.

```
    d.secure = request.uri.secure
```

The remote sequence number MUST be set to the value of the sequence number in the CSeq header field of the request. The local sequence number MUST be empty. The call identifier component of the dialog ID MUST be set to the value of the Call-ID in the request. The local tag component of the dialog ID MUST be set to the tag in the To field in the response to the request (which always includes a tag), and the remote tag component of the dialog ID MUST be set to the tag from the From field in the request. A UAS MUST be prepared to receive a request without a tag in the From field, in which case the tag is considered to have a value of null.

```
    d.localSeq, d.localSeq = 0, request.CSeq.number
    d.callId = request['Call-ID'].value
    d.localTag, d.remoteTag = response.To.tag, request.From.tag
```

The remote URI MUST be set to the URI in the From field, and the local URI MUST be set to the URI in the To field.

```
    d.localParty, d.remoteParty = Address(str(request.To.value)), Address(str(request.From.value))
    d.remoteTarget = URI(str(request.first('Contact').value.uri))
    # TODO: retransmission timer for 2xx in UAC
```

```
    stack.dialogs[d.id] = d
    return d
```

When a UAC sends a request that can establish a dialog (such as an INVITE) it MUST provide a SIP or SIPS URI with global scope (i.e., the same SIP URI can be used in messages outside this dialog) in the Contact header field of the request. If the request has a Request-URI or a topmost Route header field value with a SIPS URI, the Contact header field MUST contain a SIPS URI. When a UAC receives a response that establishes a dialog, it constructs the state of the dialog. This state MUST be maintained for the duration of the dialog.

If the request was sent over TLS, and the Request-URI contained a SIPS URI, the "secure" flag is set to TRUE.

The route set MUST be set to the list of URIs in the Record-Route header field from the response, taken in reverse order and preserving all URI parameters. If no Record-Route header field is present in the response, the route set MUST be set to the empty set. This route set, even if empty, overrides any pre-existing route set for future requests in this dialog. The remote target MUST be set to the URI from the Contact header field of the response.

The local sequence number MUST be set to the value of the sequence number in the CSeq header field of the request. The remote sequence number MUST be empty (it is established when the remote UA sends a request within the dialog). The call identifier component of the dialog ID MUST be set to the value of the Call-ID in the request. The local tag component of the dialog ID MUST be set to the tag in the From field in the request, and the remote tag component of the dialog ID MUST be set to the tag in the To field of the response. A UAC MUST be prepared to receive a response without a tag in the To field, in which case the tag is considered to have a value of null.

The remote URI MUST be set to the URI in the To field, and the local URI MUST be set to the URI in the From field.

```
    @staticmethod
    def createClient(stack, request, response, transaction):
        d = Dialog(stack, request, False)
        d.request = request
        d.routeSet = [x for x in reversed(response.all('Record-Route'))] if response['Record-Route'] else None
        d.secure = request.uri.secure
        d.localSeq, d.remoteSeq = request.CSeq.number, 0
        d.callId = request['Call-ID'].value
        d.localTag, d.remoteTag = request.From.tag, response.To.tag
        d.localParty, d.remoteParty = Address(str(request.From.value)), Address(str(request.To.value))
        d.remoteTarget = URI(str(response.first("Contact").value.uri))
        stack.dialogs[d.id] = d
        return d
```

A dialog ID is also associated with all responses and with any request that contains a tag in the To field. The rules for computing the dialog ID of a message depend on whether the SIP element is a UAC or UAS. For a UAC, the Call-ID value of the dialog ID is set to the Call-ID of the message, the remote tag is set to the tag in the To field of the message, and the local tag is set to the tag in the From field of the message (these rules apply to both requests and responses). As one would expect for a UAS, the Call-ID value of the dialog ID is set to the Call-ID of the message, the remote tag is set to the tag in the From field of the message, and the local tag is set to the tag in the To field of the message.

The method `extractId` extracts the dialog identifier string from a given incoming request or response `Message`.

```
@staticmethod
def extractId(m):
    return m['Call-ID'].value + '|' + (m.To.tag if m.method else m.From.tag) + '|' + (m.From.tag if m.method else m.To.tag)
```

The constructor takes the original request `Message`, the `server` flag indicating whether this is a UAS or UAC, and the original `transaction` reference to create the `Dialog` object out of an existing UAS or UAC.

```
def __init__(self, stack, request, server, transaction=None):
    UserAgent.__init__(self, stack, request, server) # base class method
    self.servers, self.clients = [], [] # pending server and client transactions
    self._id = None
    if transaction: transaction.app = self # this is higher layer of transaction
```

Destroying an existing dialog is done by invoking the `close` method. It removes the dialog object from the table of dialogs maintained in the stack context. TODO: we should set the `stack` property to `None`, but it causes problem in `receivedResponse` method if the `stack` is `None`.

```
def close(self):
    if self.stack:
        if self.id in self.stack.dialogs: del self.stack.dialogs[self.id]
        # self.stack = None
```

A dialog is identified at each UA with a dialog ID, which consists of a Call-ID value, a local tag and a remote tag. The dialog ID at each UA involved in the dialog is not the same. Specifically, the local tag at one UA is identical to the remote tag at the peer UA. The tags are opaque tokens that facilitate the generation of unique dialog IDs.

The id property refers to the dialog identifier string, which is constructed from the `Call-ID`, the local `tag` and the remote `tag` parameters.

```
@property
def id(self):
    if not self._id: self._id = self.callId + '|' + self.localTag + '|' + self.remoteTag
    return self._id
```

From RFC3261 p.73 – A request within a dialog is constructed by using many of the components of the state stored as part of the dialog. The tag in the To header field of the request MUST be set to the remote tag of the dialog ID.

```
def createRequest(self, method, content=None, contentType=None):
    request = UserAgent.createRequest(self, method, content, contentType)
    if self.remoteTag: request.To.tag = self.remoteTag
```

If the route set is empty, the UAC MUST place the remote target URI into the Request-URI. The UAC MUST NOT add a Route header field to the request.

If the route set is not empty, and its first URI does not contain the lr parameter, the UAC MUST place the first URI from the route set into the Request-URI, stripping any parameters that are not allowed in a Request-URI. The

UAC MUST add a Route header field containing the remainder of the route set values in order, including all parameters. The UAC MUST then place the remote target URI into the Route header field as the last value.

If the route set is not empty, and the first URI in the route set contains the lr parameter (see Section 19.1.1), the UAC MUST place the remote target URI into the Request-URI and MUST include a Route header field containing the route set values in order, including all parameters.

```
    if self.routeSet and len(self.routeSet)>0 and 'lr' not in self.routeSet[0].value.uri.param: # strict route
        request.uri = self.routeSet[0].value.uri.dup()
        del request.uri.param['lr']
    return request
```

Once the request has been constructed, the address of the server is computed and the request is sent, using the same procedures for requests outside of a dialog.

```
def createResponse(self, response, responsetext, content=None, contentType=None):
    if len(self.servers) == 0: raise ValueError, 'No server transaction to create response'
    request = self.servers[0].request
    response = Message.createResponse(response, responsetext, None, content, request)
    if contentType: response['Content-Type'] = Header(contentType, 'Content-Type')
    if response.response != 100 and 'tag' not in response.To:
        response.To.tag = self.localTag
    return response
```

To send a new response in this dialog for the first pending server transaction the application invokes the `sendResponse` method. The first argument can be either the response status code or a well formatted response `Message`.

```
def sendResponse(self, response, responsetext=None, content=None, contentType=None, createDialog=True):
    if len(self.servers) == 0: raise ValueError, 'No server transaction to send response'
    self.transaction, self.request = self.servers[0], self.servers[0].request
    UserAgent.sendResponse(self, response, responsetext, content, contentType, False)
    code = response if isinstance(response, int) else response.response
    if code >= 200:
        self.servers.pop(0) # no more pending if final response sent
```

```
def sendCancel(self):
    if len(self.clients) == 0:
        if _debug: print 'No client transaction to send cancel'
        return
    self.transaction, self.request = self.clients[0], self.clients[0].request
    UserAgent.sendCancel(self)
```

```
def receivedRequest(self, transaction, request):
    if self.remoteSeq != 0 and request.CSeq.number < self.remoteSeq:
```

```
            if _debug: print 'Dialog.receivedRequest() CSeq is old', request.CSeq.number, '<', self.remoteSeq
            self.sendResponse(500, 'Internal server error - invalid CSeq')
            return
        self.remoteSeq = request.CSeq.number

        if request.method == 'INVITE' and request.Contect:
            self.remoteTarget = request.first('Contact').value.uri.dup()

        if request.method == 'ACK' or request.method == 'CANCEL':
            self.servers = filter(lambda x: x != transaction, self.servers) # remove from pending
            if request.method == 'ACK':
                self.stack.receivedRequest(self, request)
            else:
                self.stack.cancelled(self, transaction.request)
            return

        self.servers.append(transaction) # make it pending
        self.stack.receivedRequest(self, request)
```

```
    def receivedResponse(self, transaction, response):
        '''Incoming response in a dialog.'''
        if response.is2xx and response.Contact and transaction and transaction.request.method == 'INVITE':
            self.remoteTarget = response.first('Contact').value.uri.dup()
        if not response.is1xx: # final response
            self.clients = filter(lambda x: x != transaction, self.clients) # remove from pending

        if response.response == 408 or response.response == 481: # remote doesn't recognize the dialog
            self.close()

        if response.response == 401 or response.response == 407:
            if not self.authenticate(response, transaction):
                self.stack.receivedResponse(self, response)
        elif transaction:
            self.stack.receivedResponse(self, response)

        if self.autoack and response.is2xx and (transaction and transaction.request.method == 'INVITE' or
response.CSeq.method == 'INVITE'):
            self.sendRequest(self.createRequest('ACK'))
```

Transaction

From RFC3261 p.122 -- SIP is a transactional protocol: interactions between components take place in a series of independent message exchanges. Specifically, a SIP transaction consists of a single request and any responses to that request, which include zero or more provisional responses and one or more final responses.

> Transactions have a client side and a server side. The client side is known as a client transaction and the server side as a server transaction. The client transaction sends the request, and the server transaction sends the response. The client and server transactions are logical functions that are embedded in any number of elements. Specifically, they exist within user agents and stateful proxy servers.
>
> The purpose of the client transaction is to receive a request from the element in which the client is embedded (call this element the "Transaction User" or TU; it can be a UA or a stateful proxy), and reliably deliver the request to a server transaction. The client transaction is also responsible for receiving responses and delivering them to the TU, filtering out any response retransmissions or disallowed responses (such as a response to ACK).
>
> Similarly, the purpose of the server transaction is to receive requests from the transport layer and deliver them to the TU. The server transaction filters any request retransmissions from the network. The server transaction accepts responses from the TU and delivers them to the transport layer for transmission over the network.

We define a class `Transaction` to represent a SIP transaction. This is an abstract class. The actual implementations of client and server transactions are done in `ClientTransaction` and `ServerTransaction` classes, respectively. These classes are used for non-INVITE transaction. SIP defines different processing for INVITE and non-INVITE transactions. The INVITE transactions are implemented using `InviteClientTransaction` and `InviteServerTransaction` classes. The transaction user (or TU) in our implementation is `UserAgent` object (or the derived `Dialog` object).

Transaction properties

Let's start by defining the transaction object properties. A transaction is identified by an identifier, `id`. The transaction identifier is usually derived from the `branch` parameter of the top-most `Via` header. Thus, we store the `branch` property as well. Each transaction has an original SIP `request` from which the transaction was created. The associated `transport` information and the `remote` host-port tuple give information about where to send a request or response in a transaction. We store the `tag` supplied by the TU for a server transaction. The `server` Boolean flag indicates whether this is a client (False) or server (True) transaction. The transaction module may need to use the functions from the `Stack` object referred by the `stack` property. A reference to the transaction user (TU) is stored in the `app` property. Finally, the transaction has collection of active `timers` as well as `timer` duration values for different type of timers as defined in the specification.

```
class Transaction(object):
    def __init__(self, server):
        self.branch = self.id = self.stack = self.app = self.request = self.transport = self.remote = self.tag = None
        self.server, self.timers, self.timer = server, {}, Timer()
```

When a transaction is closed, we stop all the timers and remove this transaction instance from the collection of `transactions` maintained by the `Stack`. As described earlier, the stack object maintains a table of all the transactions indexed by the transaction identifier string.

```
    def close(self):
        self.stopTimers()
        if self.stack:
            if self.id in self.stack.transactions: del self.stack.transactions[self.id]
```

Note that we couldn't use the destructor method, because as long as a reference to this transaction is stored in the transactions table, the destructor will not get invoked. Hence we need an explicit `close` method to destroy the transaction. The `close` method gets invoked whenever the transaction state is changed to

"terminating". Thus we define another property, `state`, to maintain the transaction state and explicitly invoke `close` when the state changes to "terminating".

> From RFC3261 – The client transaction MUST be destroyed the instant it enters the "Terminated" state. This is actually necessary to guarantee correct operation.
>
> Once the transaction is in the "Terminated" state, it MUST be destroyed immediately. As with client transactions, this is needed to ensure reliability of the 2xx responses to INVITE.

```
def state():
    def fset(self, value):
        self._state = value
        if self._state == 'terminating': self.close() # automatically close when state goes terminating
    def fget(self): return self._state
    return locals()
state = property(**state())
```

SIP defines four headers – `To`, `From`, `CSeq`, `Call-ID` – as transaction identifying headers. The values of these header fields remains the same within a transaction, although the header parameters may change – e.g., the `tag` parameter gets added to the `To` header. We define a read-only property, `headers`, which gives a list of these four headers.

```
@property
def headers(self):
    return map(lambda x: self.request[x], ['To', 'From', 'CSeq', 'Call-ID'])
```

Creating branch and transaction identifiers

SIP imposes certain restrictions on creation of `branch` parameter. In particular, the RFC3261 compliant implementation must start the branch parameter with "z9hG4bK" to distinguish against previous RFC2543 implementations. In practice, an implementation must choose the branch parameter carefully, so that it can be used to match a transaction, i.e., act as a transaction identifier. Most of the implementations that I have seen use some combination of transaction headers to create the branch parameter.

> From RFC3261 p.29 – The Via header field value MUST contain a branch parameter. This parameter is used to identify the transaction created by that request. This parameter is used by both the client and the server.
>
> The branch ID inserted by an element compliant with this specification MUST always begin with the characters "z9hG4bK". These 7 characters are used as a magic cookie (7 is deemed sufficient to ensure that an older RFC 2543 implementation would not pick such a value), so that servers receiving the request can determine that the branch ID was constructed in the fashion described by this specification (that is, globally unique). Beyond this requirement, the precise format of the branch token is implementation-defined.

The function `createBranch` defined below uses the information from the transaction identifying headers, along with the `server` flag. The `server` flag is needed so that a client transaction doesn't interfere with a server transaction while searching for a transaction in the transactions table. Note that we only use the header value of `To` and `From` without the parameters, and the `number` field from `CSeq` header without the `method` name. This is important so that a response with `tag` parameter in the `To` header gets matched with the original transaction that didn't have the `tag` parameter in the `To` header. Secondly, the `branch` parameter for the CANCEL and ACK request remains the same as that of the original INVITE if we don't include the `method` of `CSeq` header in computing the `branch`. Finally we use a one-way hash such as MD5 and modified Base64 encoding to construct a random-looking `branch` parameter from the assembled

data. The modified Base64 encoding is needed because certain characters in the original Base64 grammar are not allowed in the `branch` grammar by the specification.

```
from hashlib import md5
from base64 import urlsafe_b64encode
...
   @staticmethod
   def createBranch(request, server):
      '''Static method to create a branch parameter from request (Message) and server (Boolean)
      or using [To, From, Call-ID, CSeq-number(int)] and server (Boolean).'''
      To, From, CallId, CSeq = (request.To.value, request.From.value, request['Call-ID'].value, request.CSeq.number) if
isinstance(request, Message) else (request[0], request[1], request[2], request[3])
      data = str(To).lower() + '|' + str(From).lower() + '|' + str(CallId) + '|' + str(CSeq) + '|' + str(server)
      return 'z9hG4bK' + str(urlsafe_b64encode(md5(data).digest())).replace('=','.')
```

For added flexibility, we allow overloaded method invocation where the first argument can be either a Message object or a list of the individual fields needed for computing the branch.

```
Transaction.createBranch(request, True) # create a branch from request for a server transaction
Transaction.createBranch([m.To.value, m.From.value, m['Call-ID'].value, m.CSeq.number], False) # use individual fields
```

> The branch parameter value MUST be unique across space and time for all requests sent by the UA. The exceptions to this rule are CANCEL and ACK for non-2xx responses. A CANCEL request will have the same value of the branch parameter as the request it cancels. An ACK for a non-2xx response will also have the same branch ID as the INVITE whose response it acknowledges.

The uniqueness property of the branch parameter allows us to use it as the transaction identifier, with a couple of exception – if the method is either CANCEL or ACK, then even though the `branch` parameter is same as the original INVITE request, the transaction identifier should be different. We define the `createId` method to construct such a transaction identifier, which appends the method name if the method is ACK or CANCEL. The transaction identifier is used as a key in our lookup table of transactions.

```
   @staticmethod
   def createId(branch, method):
      return branch if method != 'ACK' and method != 'CANCEL' else branch + '|' + method
```

Creating a transaction

The TU wishes to create a new server transaction it invokes the `createServer` factory method by supplying the incoming request `Message`, the associated transport information on which the request was received and the application `tag` parameter to use in the transaction response. The method hides the implementation details of what type of object is used for the particular transaction, e.g., whether INVITE or non-INVITE transactions use separate implementations.

```
   @staticmethod
```

```
    def createServer(stack, app, request, transport, tag):
        t = request.method == 'INVITE' and InviteServerTransaction() or ServerTransaction()
        t.stack, t.app, t.request, t.transport, t.tag = stack, app, request, transport, tag
```

For a server transaction, certain transaction properties are derived from the incoming message, e.g., the `branch` and `remote` address property are extracted from the top-most `Via` header. If a `branch` parameter is missing in the request, probably due to old implementation of the specification, then the `branch` parameter is constructed using the request message as described earlier.

```
        t.remote = request.first('Via').viaUri.hostPort
        t.branch = request.first('Via').branch if request.Via != None and 'branch' in request.first('Via') else
Transaction.createBranch(request, True)
```

Finally, the transaction identifier is created, the transaction is stored in the transactions table, the transaction state machine is started, and the `transaction` object is returned as the newly created server transaction.

```
        t.id = Transaction.createId(t.branch, request.method)
        stack.transactions[t.id] = t
        t.start()
        return t
```

> From RFC3261 – The TU communicates with the client transaction through a simple interface. When the TU wishes to initiate a new transaction, it creates a client transaction and passes it the SIP request to send and an IP address, port, and transport to which to send it. The client transaction begins execution of its state machine. Valid responses are passed up to the TU from the client transaction.
>
> There are two types of client transaction state machines, depending on the method of the request passed by the TU. One handles client transactions for INVITE requests. This type of machine is referred to as an INVITE client transaction. Another type handles client transactions for all requests except INVITE and ACK. This is referred to as a non-INVITE client transaction.

When the TU wishes to create a client transaction for sending out a new request, it uses the `createClient` method and supplies the request `Message`, the associated transport which will be used to send the request and the `remote` host-port tuple to which we want to send the request to. Similar to the creation of server transaction, this method also hides the implementation details of the type of transaction object created. The rest of the processing is very similar to the previous method.

```
    @staticmethod
    def createClient(stack, app, request, transport, remote):
        t = request.method == 'INVITE' and InviteClientTransaction() or ClientTransaction()
        t.stack, t.app, t.request, t.remote, t.transport = stack, app, request, remote, transport
        t.branch = request.first('Via').branch if request.Via != None and 'branch' in request.first('Via') else
Transaction.createBranch(request, False)
        t.id = Transaction.createId(t.branch, request.method)
        stack.transactions[t.id] = t
        t.start()
```

```
        return t
```

Comparing a request against a transaction

The `equals` method on the transaction is used by the `Stack.findOtherTransaction` method to check whether a request `r` matches an existing transaction `t1`, such that transaction `t1` is different from original transaction `t2`, but has the same direction (client or server) as the original transaction `t2`. When an incoming request matches another transaction (`t1`) even though the request is part of another original transaction (`t2`), we have a request merging situation, hence the request should get rejected.

```
    @staticmethod
    def equals(t1, r, t2):
        t = t1.request
        a = r.To.value.uri == t.To.value.uri
        a = a and (r.From.value.uri == t.From.value.uri)
        a = a and (r['Call-ID'].value == t['Call-ID'].value)
        a = a and (r.CSeq.value == t.CSeq.value)
        a = a and (r.From.tag == t.From.tag)
        a = a and (t2.server == t1.server)
        return a
```

Creating ACK, CANCEL and responses

To create an ACK request in a client transaction, we use the original request URI of the transaction with transaction identifying headers. The method returns `None` for server transaction.

```
    def createAck(self):
        return Message.createRequest('ACK', str(self.request.uri), self.headers) if self.request and not self.server else None
```

To create a CANCEL request in a client transaction, we again use the original request URI of the transaction with transaction identifying headers. Additionally, for a CANCEL request the `Route` header is copied from the original request if needed, and only one `Via` header, i.e., top-most one, is kept in the request. The method returns `None` for server transaction.

```
    def createCancel(self):
        m = Message.createRequest('CANCEL', str(self.request.uri), self.headers) if self.request and not self.server else
None
        if m and self.request.Route: m.Route = self.request.Route
        if m: m.Via = self.request.first('Via') # only top Via included
        return m
```

To create a response in a server transaction, we use the original request and add the response status code (`response`) and reason phrase (`responsetext`). If the response is not “100 Trying” then we also add

the `tag` parameter in the `To` header if one is missing. TODO: this should be moved to UAS? The method returns `None` for client transaction.

```
    def createResponse(self, response, responsetext):
        m = Message.createResponse(response, responsetext, None, None, self.request) if self.request and self.server else
None
        if response != 100 and 'tag' not in m.To: m.To['tag'] = self.tag
        return m
```

Transaction timers

In the transaction state machine, several timers exist. The `startTimer` method is used to start a new named timer for the given timeout duration. To create a timer if it doesn't already exist in this transaction, we invoke the application callback `createTimer`.

```
    def startTimer(self, name, timeout):
        if timeout > 0:
            if name in self.timers:
                timer = self.timers[name]
            else:
                timer = self.timers[name] = self.stack.createTimer(self)
            timer.delay = timeout
            timer.start()
```

When the timer expires, we invoke the `timeout` handler method which is implemented for individual transaction state machines. The `timedout` method is invoked by the actual timer implementation of the application, the one that was returned in `createTimer`.

```
    def timedout(self, timer):
        if timer.running: timer.stop()
        found = filter(lambda x: self.timers[x] == timer, self.timers.keys())
        if len(found):
            for f in found: del self.timers[f]
            self.timeout(found[0], timer.delay)
```

When cleaning up a transaction, we may need to stop all the timers associated with this transaction. The `stopTimers` method can be used for that purpose.

```
    def stopTimers(self):
        for v in self.timers.values(): v.stop()
        del self.timers
```

Timers

RFC3261 defines several timers which we abstract out in our implementation of the `Timer` class. In particular, the named timers T1, T2 and T4 configure the timeout values of all other timers, timer A to K. The default values of T1, T2 and T4 times are 500, 4000 and 5000 milliseconds. If a different default value is needed then the transaction can create the `Timer` object with those different values during construction.

```
class Timer(object):
    def __init__(self, T1=500, T2=4000, T4=5000):
        self.T1, self.T2, self.T4 = T1, T2, T4
```

Timer A's value is initially same as T1, and gets updated every time the timer expires.

```
    def A(self): return self.T1
```

Timer B's value is 64×T1.

```
    def B(self): return 64*self.T1
```

TODO: why no timer C?

Timer D's value is also similar to timer B, except that it caps at 32 seconds.

```
    def D(self): return max(64*self.T1, 32000)
```

Timer I's value is same as that of timer T4.

```
    def I(self): return self.T4
```

Finally we turn these derived timer values into read-only properties, such that initial values of timers A, E, G are all same, timers B, F, H, J are all same, and timers I and K are same.

```
    A, B, D, E, F, G, H, I, J, K = map(lambda x: property(x), [A, B, D, A, B, A, B, I, B, I])
```

INVITE client transaction

> From RFC3261 p.125 – The INVITE transaction consists of a three-way handshake. The client transaction sends an INVITE, the server transaction sends responses, and the client transaction sends an ACK. For unreliable transports (such as UDP), the client transaction retransmits requests at an interval that starts at T1 seconds and doubles after every retransmission. T1 is an estimate of the round-trip time (RTT), and it defaults to 500 ms. Nearly all of the transaction timers described here scale with T1, and changing T1 adjusts their values. The request is not retransmitted over reliable transports. After receiving a 1xx response, any retransmissions cease altogether, and the client waits for further responses. The server transaction can send additional 1xx

responses, which are not transmitted reliably by the server transaction. Eventually, the server transaction decides to send a final response. For unreliable transports, that response is retransmitted periodically, and for reliable transports, it is sent once. For each final response that is received at the client transaction, the client transaction sends an ACK, the purpose of which is to quench retransmissions of the response.

```
class InviteClientTransaction(Transaction):
  def __init__(self):
    Transaction.__init__(self, False)
```

The initial state, "calling", MUST be entered when the TU initiates a new client transaction with an INVITE request. The client transaction MUST pass the request to the transport layer for transmission (see Section 18). If an unreliable transport is being used, the client transaction MUST start timer A with a value of T1. If a reliable transport is being used, the client transaction SHOULD NOT start timer A (Timer A controls request retransmissions). For any transport, the client transaction MUST start timer B with a value of 64*T1 seconds (Timer B controls transaction timeouts).

```
  def start(self):
    self.state = 'calling'
    if not self.transport.reliable:
      self.startTimer('A', self.timer.A)
    self.startTimer('B', self.timer.B)
    self.stack.send(self.request, self.remote, self.transport)
```

If the client transaction receives a provisional response while in the "Calling" state, it transitions to the "Proceeding" state. In the "Proceeding" state, the client transaction SHOULD NOT retransmit the request any longer. Furthermore, the provisional response MUST be passed to the TU. Any further provisional responses MUST be passed up to the TU while in the "Proceeding" state.

```
  def receivedResponse(self, response):
    if response.is1xx:
      if self.state == 'calling':
        self.state = 'proceeding'
        self.app.receivedResponse(self, response)
      elif self.state == 'proceeding':
        self.app.receivedResponse(self, response)
```

When in either the "Calling" or "Proceeding" states, reception of a 2xx response MUST cause the client transaction to enter the "Terminated" state, and the response MUST be passed up to the TU. The handling of this response depends on whether the TU is a proxy core or a UAC core. A UAC core will handle generation of the ACK for this response, while a proxy core will always forward the 200 (OK) upstream. The differing treatment of 200 (OK) between proxy and UAC is the reason that handling of it does not take place in the transaction layer.

```
    elif response.is2xx:
      if self.state == 'calling' or self.state == 'proceeding':
        self.state = 'terminated'
        self.app.receivedResponse(self, response)
```

When in either the "Calling" or "Proceeding" states, reception of a response with status code from 300-699 MUST cause the client transaction to transition to "Completed". The client transaction MUST pass the received response up to the TU, and the client transaction MUST generate an ACK request, even if the transport is reliable (guidelines for constructing the ACK from the response are given in Section 17.1.1.3) and then pass the ACK to the transport layer for transmission. The ACK MUST be sent to the same address, port, and transport to which the original request was sent. The client transaction SHOULD start timer D when it enters the "Completed" state, with a value of at least 32 seconds for unreliable transports, and a value of zero seconds for reliable transports. Timer D reflects the amount of time that the server transaction can remain in the "Completed" state when unreliable transports are used. This is equal to Timer H in the INVITE server transaction, whose default is 64*T1. However, the client transaction does not know the value of T1 in use by the server transaction, so an absolute minimum of 32s is used instead of basing Timer D on T1.

```
    else: # failure
        if self.state == 'calling' or self.state == 'proceeding':
            self.state = 'completed'
            self.stack.send(self.createAck(response), self.remote, self.transport)
            self.app.receivedResponse(self, response)
            if not self.transport.reliable:
                self.startTimer('D', self.timer.D)
            else:
                self.timeout('D', 0)
```

Any retransmissions of the final response that are received while in the "Completed" state MUST cause the ACK to be re-passed to the transport layer for retransmission, but the newly received response MUST NOT be passed up to the TU.

```
        elif self.state == 'completed':
            self.stack.send(self.createAck(response), self.remote, self.transport)
```

When timer A fires, the client transaction MUST retransmit the request by passing it to the transport layer, and MUST reset the timer with a value of 2*T1. The formal definition of retransmit within the context of the transaction layer is to take the message previously sent to the transport layer and pass it to the transport layer once more.

When timer A fires 2*T1 seconds later, the request MUST be retransmitted again (assuming the client transaction is still in this state). This process MUST continue so that the request is retransmitted with intervals that double after each transmission. These retransmissions SHOULD only be done while the client transaction is in the "calling" state.

```
def timeout(self, name, timeout):
    if self.state == 'calling':
        if name == 'A':
            self.startTimer('A', 2*timeout)
            self.stack.send(self.request, self.remote, self.transport)
```

If the client transaction is still in the "Calling" state when timer B fires, the client transaction SHOULD inform the TU that a timeout has occurred. The client transaction MUST NOT generate an ACK. The value of 64*T1 is equal to the amount of time required to send seven requests in the case of an unreliable transport.

```
        elif name == 'B':
```

```
            self.state = 'terminated'
            self.app.timeout(self)
```

If timer D fires while the client transaction is in the "Completed" state, the client transaction MUST move to the terminated state.

```
    elif self.state == 'completed':
        if name == 'D':
            self.state = 'terminated'
```

Any transport error causes the state machine to move to the “terminated” state and updates the TU with the error message.

```
def error(self, error):
    if self.state == 'calling' or self.state == 'completed':
        self.state = 'terminated'
        self.app.error(self, error)
```

The ACK request constructed by the client transaction MUST contain values for the Call-ID, From, and Request-URI that are equal to the values of those header fields in the request passed to the transport by the client transaction (call this the "original request").

```
def createAck(self, response):
    if not self.request: raise ValueError, 'No transaction request found'
    m = Message.createRequest('ACK', str(self.request.uri))
    m['Call-ID'] = self.request['Call-ID']
    m.From   = self.request.From
```

The To header field in the ACK MUST equal the To header field in the response being acknowledged, and therefore will usually differ from the To header field in the original request by the addition of the tag parameter.

```
    m.To     = response.To if response else self.request.To
```

The ACK MUST contain a single Via header field, and this MUST be equal to the top Via header field of the original request.

```
    m.Via    = self.request.first("Via") # only top Via
```

The CSeq header field in the ACK MUST contain the same value for the sequence number as was present in the original request, but the method parameter MUST be equal to "ACK".

```
    m.CSeq   = Header(str(self.request.CSeq.number) + ' ACK', 'CSeq')
```

If the INVITE request whose response is being acknowledged had Route header fields, those header fields MUST appear in the ACK. This is to ensure that the ACK can be routed properly through any downstream stateless proxies.

```
    if self.request.Route: m.Route = self.request.Route
    return m;
```

INVITE server transaction

When a server transaction is constructed for a request, it enters the "Proceeding" state. The server transaction MUST generate a 100 (Trying) response unless it knows that the TU will generate a provisional or final response within 200 ms, in which case it MAY generate a 100 (Trying) response. This provisional response is needed to quench request retransmissions rapidly in order to avoid network congestion. The request MUST be passed to the TU.

```
class InviteServerTransaction(Transaction):
  def __init__(self):
    Transaction.__init__(self, True)
  def start(self):
    self.state = 'proceeding'
    self.sendResponse(self.createResponse(100, 'Trying'))
    self.app.receivedRequest(self, self.request)
```

Furthermore, while in the "Completed" state, if a request retransmission is received, the server SHOULD pass the response to the transport for retransmission.

```
  def receivedRequest(self, request):
    if self.request.method == request.method: # retransmitted
      if self.state == 'proceeding' or self.state == 'completed':
        self.stack.send(self.lastResponse, self.remote, self.transport)
```

If an ACK is received while the server transaction is in the "Completed" state, the server transaction MUST transition to the "Confirmed" state. As Timer G is ignored in this state, any retransmissions of the response will cease.

The purpose of the "Confirmed" state is to absorb any additional ACK messages that arrive, triggered from retransmissions of the final response. When this state is entered, timer I is set to fire in T4 seconds for unreliable transports, and zero seconds for reliable transports.

```
    elif request.method == 'ACK':
      if self.state == 'completed':
        self.state = 'confirmed'
        if not self.transport.reliable:
          self.startTimer('I', self.timer.I)
        else:
          self.timeout('I', 0)
      elif self.state == 'confirmed':
```

```
            pass  # ignore the retransmitted ACK
```

If timer G fires, the response is passed to the transport layer once more for retransmission, and timer G is set to fire in MIN(2*T1, T2) seconds. From then on, when timer G fires, the response is passed to the transport again for transmission, and timer G is reset with a value that doubles, unless that value exceeds T2, in which case it is reset with the value of T2.

```
def timeout(self, name, timeout):
    if self.state == 'completed':
        if name == 'G':
            self.startTimer('G', min(2*timeout, self.timer.T2))
            self.stack.send(self.lastResponse, self.remote, self.transport)
```

If timer H fires while in the "Completed" state, it implies that the ACK was never received. In this case, the server transaction MUST transition to the "Terminated" state, and MUST indicate to the TU that a transaction failure has occurred.

```
        elif name == 'H':
            self.state = 'terminated'
            self.app.timeout(self)
```

Once timer I fires, the server MUST transition to the "Terminated" state.

```
    elif self.state == 'confirmed':
        if name == 'I':
            self.state = 'terminated'
```

As with the client transaction, any transport error is treated as error and propagated to the TU.

```
def error(self, error):
    if self.state == 'proceeding' or self.state == 'confirmed':
        self.state = 'terminated'
        self.app.error(self, error)
```

The TU passes any number of provisional responses to the server transaction. So long as the server transaction is in the "Proceeding" state, each of these MUST be passed to the transport layer for transmission. They are not sent reliably by the transaction layer (they are not retransmitted by it) and do not cause a change in the state of the server transaction. If a request retransmission is received while in the "Proceeding" state, the most recent provisional response that was received from the TU MUST be passed to the transport layer for retransmission.

```
def sendResponse(self, response):
    self.lastResponse = response
    if response.is1xx:
        if self.state == 'proceeding':
            self.stack.send(response, self.remote, self.transport)
```

If, while in the "Proceeding" state, the TU passes a 2xx response to the server transaction, the server transaction MUST pass this response to the transport layer for transmission. It is not retransmitted by the server transaction; retransmissions of 2xx responses are handled by the TU. The server transaction MUST then transition to the "Terminated" state.

```
    elif response.is2xx:
        if self.state == 'proceeding':
            self.state = 'terminated'
            self.stack.send(response, self.remote, self.transport)
```

While in the "Proceeding" state, if the TU passes a response with status code from 300 to 699 to the server transaction, the response MUST be passed to the transport layer for transmission, and the state machine MUST enter the "Completed" state. For unreliable transports, timer G is set to fire in T1 seconds, and is not set to fire for reliable transports.

```
    else: # failure
        if self.state == 'proceeding':
            self.state = 'completed'
            if not self.transport.reliable:
                self.startTimer('G', self.timer.G)
```

When the "Completed" state is entered, timer H MUST be set to fire in 64*T1 seconds for all transports. Timer H determines when the server transaction abandons retransmitting the response. Its value is chosen to equal Timer B, the amount of time a client transaction will continue to retry sending a request.

```
            self.startTimer('H', self.timer.H)
            self.stack.send(response, self.remote, self.transport)
```

Non-INVITE client transaction

From RFC3261 p.130 – Non-INVITE transactions do not make use of ACK. They are simple request-response interactions. For unreliable transports, requests are retransmitted at an interval which starts at T1 and doubles until it hits T2. If a provisional response is received, retransmissions continue for unreliable transports, but at an interval of T2. The server transaction retransmits the last response it sent, which can be a provisional or final response, only when a retransmission of the request is received. This is why request retransmissions need to continue even after a provisional response; they are to ensure reliable delivery of the final response. Unlike an INVITE transaction, a non-INVITE transaction has no special handling for the 2xx response. The result is that only a single 2xx response to a non-INVITE is ever delivered to a UAC.

```
class ClientTransaction(Transaction):
    def __init__(self):
        Transaction.__init__(self, False)
```

The "Trying" state is entered when the TU initiates a new client transaction with a request. When entering this state, the client transaction SHOULD set timer F to fire in 64*T1 seconds. The request MUST be passed to the

transport layer for transmission. If an unreliable transport is in use, the client transaction MUST set timer E to fire in T1 seconds.

```
def start(self):
    self.state = 'trying'
    if not self.transport.reliable:
        self.startTimer('E', self.timer.E)
    self.startTimer('F', self.timer.F)
    self.stack.send(self.request, self.remote, self.transport)
```

If a provisional response is received while in the "Trying" state, the response MUST be passed to the TU, and then the client transaction SHOULD move to the "Proceeding" state.

```
def receivedResponse(self, response):
    if response.is1xx:
        if self.state == 'trying':
            self.state = 'proceeding'
            self.app.receivedResponse(self, response)
        elif self.state == 'proceeding':
            self.app.receivedResponse(self, response)
```

If a final response (status codes 200-699) is received while in the "Trying" state, the response MUST be passed to the TU, and the client transaction MUST transition to the "Completed" state.

If a final response (status codes 200-699) is received while in the "Proceeding" state, the response MUST be passed to the TU, and the client transaction MUST transition to the "Completed" state.

```
    elif response.isfinal:
        if self.state == 'trying' or self.state == 'proceeding':
            self.state = 'completed'
            self.app.receivedResponse(self, response)
```

Once the client transaction enters the "Completed" state, it MUST set Timer K to fire in T4 seconds for unreliable transports, and zero seconds for reliable transports.

```
            if not self.transport.reliable:
                self.startTimer('K', self.timer.K)
            else:
                self.timeout('K', 0)
```

If timer E fires while still in this state, the timer is reset, but this time with a value of MIN(2*T1, T2). When the timer fires again, it is reset to a MIN(4*T1, T2). This process continues so that retransmissions occur with an exponentially increasing interval that caps at T2. The default value of T2 is 4s, and it represents the amount of time a non-INVITE server transaction will take to respond to a request, if it does not respond immediately. For the default values of T1 and T2, this results in intervals of 500 ms, 1 s, 2 s, 4 s, 4 s, 4 s, etc.

If Timer E fires while in the "Proceeding" state, the request MUST be passed to the transport layer for retransmission, and Timer E MUST be reset with a value of T2 seconds.

```
def timeout(self, name, timeout):
    if self.state == 'trying' or self.state == 'proceeding':
        if name == 'E':
            timeout = min(2*timeout, self.timer.T2) if self.state == 'trying' else self.timer.T2
            self.startTimer('E', timeout)
            self.stack.send(self.request, self.remote, self.transport)
```

If Timer F fires while the client transaction is still in the "Trying" state, the client transaction SHOULD inform the TU about the timeout, and then it SHOULD enter the "Terminated" state.

If timer F fires while in the "Proceeding" state, the TU MUST be informed of a timeout, and the client transaction MUST transition to the terminated state.

```
        elif name == 'F':
            self.state = 'terminated'
            self.app.timeout(self)
```

If Timer K fires while in this (“completed”) state, the client transaction MUST transition to the "Terminated" state.

```
    elif self.state == 'completed':
        if name == 'K':
            self.state = 'terminated'
```

The client transaction SHOULD inform the TU that a transport failure has occurred, and the client transaction SHOULD transition directly to the "Terminated" state.

```
def error(self, error):
    if self.state == 'trying' or self.state == 'proceeding':
        self.state = 'terminated'
        self.app.error(self, error)
```

Non-INVITE server transaction

From RFC3261 p.137 – The state machine is initialized in the "Trying" state and is passed a request other than INVITE or ACK when initialized. This request is passed up to the TU.

```
class ServerTransaction(Transaction):
    def __init__(self):
        Transaction.__init__(self, True)
    def start(self):
        self.state = 'trying'
        self.app.receivedRequest(self, self.request)
```

If a retransmission of the request is received while in the "Proceeding" state, the most recently sent provisional response MUST be passed to the transport layer for retransmission.

While in the "Completed" state, the server transaction MUST pass the final response to the transport layer for retransmission whenever a retransmission of the request is received.

```
def receivedRequest(self, request):
    if self.request.method == request.method: # retransmitted
        if self.state == 'proceeding' or self.state == 'completed':
            self.stack.send(self.lastResponse, self.remote, self.transport)
```

Once in the "Trying" state, any further request retransmissions are discarded.

```
        elif self.state == 'trying':
            pass # just ignore the retransmitted request
```

The server transaction remains in this state until Timer J fires, at which point it MUST transition to the "Terminated" state.

```
def timeout(self, name, timeout):
    if self.state == 'completed':
        if name == 'J':
            self.state = 'terminated'
```

As with the client transaction, a transport error is propagated up the TU and the state transitions to “terminated”.

```
def error(self, error):
    if self.state == 'completed':
        self.state = 'terminated'
        self.app.error(self, error)
```

While in the "Trying" state, if the TU passes a provisional response to the server transaction, the server transaction MUST enter the "Proceeding" state. The response MUST be passed to the transport layer for transmission. Any further provisional responses that are received from the TU while in the "Proceeding" state MUST be passed to the transport layer for transmission.

```
def sendResponse(self, response):
    self.lastResponse = response;
    if response.is1xx:
        if self.state == 'trying' or self.state == 'proceedings':
            self.state = 'proceeding'
            self.stack.send(response, self.remote, self.transport)
```

If the TU passes a final response (status codes 200-699) to the server while in the "Proceeding" state, the transaction MUST enter the "Completed" state, and the response MUST be passed to the transport layer for transmission.

Any other final responses passed by the TU to the server transaction MUST be discarded while in the "Completed" state.

```
    elif response.isfinal:
```

```
        if self.state == 'proceeding' or self.state == 'trying':
            self.state = 'completed'
            self.stack.send(response, self.remote, self.transport)
```

When the server transaction enters the "Completed" state, it MUST set Timer J to fire in 64*T1 seconds for unreliable transports, and zero seconds for reliable transports.

```
            if not self.transport.reliable:
                self.startTimer('J', self.timer.J)
            else:
                self.timeout('J', 0)
```

Session description

Implementing offer-answer and SDP as per RFC 3264, RFC 4566

The Session Description Protocol (SDP) is specified in RFC 4566 and defines the format for describing the session parameters in a SIP session. In particular, the SIP INVITE request and the 2xx-class response to the INVITE request can contain the message body in SDP format. The SDP data advertises the media types, list of codecs and transport addresses for the sender. Secondly, RFC 3264 defines how a SIP user agent can offer and answer the session negotiation parameters with the help of SDP. In particular, it adds additional constraints on the base SDP for usage in a SIP telephony environment.

In this chapter we implement the modules named `rfc4566` and `rfc3264` to implement these session description and negotiation functions for SIP telephony.

Session Description Protocol (SDP)

> From RFC4566 p.3 – When initiating multimedia teleconferences, voice-over-IP calls, streaming video, or other sessions, there is a requirement to convey media details, transport addresses, and other session description metadata to the participants.
>
> SDP provides a standard representation for such information, irrespective of how that information is transported. SDP is purely a format for session description -- it does not incorporate a transport protocol, and it is intended to use different transport protocols as appropriate, including the Session Announcement Protocol, Session Initiation Protocol, Real Time Streaming Protocol, electronic mail using the MIME extensions, and the Hypertext Transport Protocol.
>
> SDP is intended to be general purpose so that it can be used in a wide range of network environments and applications. However, it is not intended to support negotiation of session content or media encodings: this is viewed as outside the scope of session description.

SDP is also used in conjunction with other protocols such as Session Announcement Protocol (SAP) and Real Time Streaming Protocol (RTSP), but those are beyond the scope of current discussion.

> From RFC4566 p.7 – An SDP session description is entirely textual using the ISO 10646 character set in UTF-8 encoding. SDP field names and attribute names use only the US-ASCII subset of UTF-8, but textual fields and attribute values MAY use the full ISO 10646 character set. Field and attribute values that use the full UTF-8 character set are never directly compared, hence there is no requirement for UTF-8 normalisation. The textual form, as opposed to a binary encoding such as ASN.1 or XDR, was chosen to enhance portability, to enable a variety of transports to be used, and to allow flexible, text-based toolkits to be used to generate and process session descriptions.

Usage

Before we jump into the implementation, let's understand the basic usage of the module `rfc4566`. We will define a class named `SDP` to represent an SDP packet. SDP is a text-based protocol. An example SDP description from RFC4566 p.10 is shown below:

```
v=0
o=jdoe 2890844526 2890842807 IN IP4 10.47.16.5
s=SDP Seminar
i=A Seminar on the session description protocol
u=http://www.example.com/seminars/sdp.pdf
```

```
e=j.doe@example.com (Jane Doe)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 99
a=rtpmap:99 h263-1998/90000
```

To implement the SDP class we first need to define how we intend to use the class. An object with dynamic properties that can be assessed either as attribute or container access forms a good programming interface.

```
>>> s = SDP("v=0\r\no=jdoe 2890844526 2890842807 IN IP4 10.47.16.5\r\ns=-\r\nc=IN IP4 224.2.17.12/127")
>>> s.i = "A Seminar on the session description protocol"
>>> s[s] = "SDP Seminar"
>>> print s.c.address, s.c.ttl
224.2.17.12 127
>>> print s.t == None
True
```

Container and attribute access

We define the `attrs` class that implements such an attribute plus container interface for accessing the various headers in the SDP. Unlike the attribute access on a regular Python object, an `attrs` object returns `None` for a missing element instead of throwing an error. This helps the programmer in writing clean source code.

```
class attrs(object):
    def __init__(self, **kwargs):
        for n,v in kwargs.items(): self[n] = v
    def __getattr__(self, name): return self.__getitem__(name)
    def __getitem__(self, name): return self.__dict__.get(name, None)
    def __setitem__(self, name, value): self.__dict__[name] = value
    def __contains__(self, name): return name in self.__dict__
```

Then we derive the SDP class from this `attrs` class to extend the additional specific attributes such as connection line.

```
class SDP(attrs):
    _multiple = 'tramb'    # header names that can appear multiple times.
    def __init__(self, value=None):
        if value:  self._parse(value)
```

Certain attributes such as "t=", "r=", etc. can appear multiple times in SDP and need to be identified separately as done by the `_multiple` property of the SDP class.

From RFC4566 p.8 – Some lines in each description are REQUIRED and some are OPTIONAL, but all MUST appear in exactly the order given here (the fixed order greatly enhances error detection and allows for a simple parser).

Before defining the parsing of the full SDP data, let's define the individual specific headers that can be represented using more than just a string.

Origin data

From RFC4566 p.11 – Origin (o=)

o=<username> <sess-id> <sess-version> <nettype> <addrtype> <unicast-address>

The "o=" field gives the originator of the session (her username and the address of the user's host) plus a session identifier and version number:

<username> is the user's login on the originating host, or it is "-" if the originating host does not support the concept of user IDs. The <username> MUST NOT contain spaces.

<sess-id> is a numeric string such that the tuple of <username>, <sess-id>, <nettype>, <addrtype>, and <unicast-address> forms a globally unique identifier for the session. The method of <sess-id> allocation is up to the creating tool, but it has been suggested that a Network Time Protocol (NTP) format timestamp be used to ensure uniqueness.

<sess-version> is a version number for this session description. Its usage is up to the creating tool, so long as <sess-version> is increased when a modification is made to the session data. Again, it is RECOMMENDED that an NTP format timestamp is used.

<nettype> is a text string giving the type of network. Initially "IN" is defined to have the meaning "Internet", but other values MAY be registered in the future.

<addrtype> is a text string giving the type of the address that follows. Initially "IP4" and "IP6" are defined, but other values MAY be registered in the future.

<unicast-address> is the address of the machine from which the session was created. For an address type of IP4, this is either the fully qualified domain name of the machine or the dotted-decimal representation of the IP version 4 address of the machine. For an address type of IP6, this is either the fully qualified domain name of the machine or the compressed textual representation of the IP version 6 address of the machine. For both IP4 and IP6, the fully qualified domain name is the form that SHOULD be given unless this is unavailable, in which case the globally unique address MAY be substituted. A local IP address MUST NOT be used in any context where the SDP description might leave the scope in which the address is meaningful (for example, a local address MUST NOT be included in an application-level referral that might leave the scope).

In general, the "o=" field serves as a globally unique identifier for this version of this session description, and the subfields excepting the version taken together identify the session irrespective of any modifications.

Let's define the `originator` class to represent the "o=" line and derive it from the `attrs` class so that it can also have dynamic attributes. The individual properties such as `usename` (str), `sessionid` (long), `version` (long), `nettype` (str), `addrtype` (str), `address` (str) are as described above. There are two methods of importance: the constructor `__init__` which is used to parse the SDP line, and the string representation method `__repr__` for format the SDP line.

```
import socket, time
...
   class originator(attrs):
      def __init__(self, value=None):
```

If a value is supplied in the constructor it parses the SDP line into individual properties by splitting the value across white-space.

```
        if value:
            self.username, self.sessionid, self.version, self.nettype, self.addrtype, self.address = value.split(' ')
            self.sessionid = int(self.sessionid)
            self.version   = int(self.version)
```

Otherwise if the value is not supplied in the constructor it assumes default values for the properties. For example, the `address` assumes local hostname or IP address, `username` is '-', `sessionid` and `version` are derived from the local time so that they are monotonically increasing, `nettype` and `addrtype` take the defaults 'IN' and 'IP4'.

```
        else:
            hostname = socket.gethostname(); ipaddress = socket.gethostbyname(hostname)
            self.username, self.sessionid, self.version, self.nettype, self.addrtype, self.address = \
            '-', int(time.time()), int(time.time()), 'IN', 'IP4', (hostname.find('.')>0 and hostname or  ipaddress)
```

Converting an object of type `originator` into a string is straightforward – just join all the properties in the right order using white-space.

```
    def __repr__(self):
        return ' '.join(map(lambda x: str(x), [self.username, self.sessionid, self.version, self.nettype, self.addrtype,
self.address]))
```

Connection data

From RFC4566 p.14 – Connection Data ("c=")

c=<nettype> <addrtype> <connection-address>

The "c=" field contains connection data.

A session description MUST contain either at least one "c=" field in each media description or a single "c=" field at the session level. It MAY contain a single session-level "c=" field and additional "c=" field(s) per media description, in which case the per-media values override the session-level settings for the respective media.

The first sub-field ("<nettype>") is the network type, which is a text string giving the type of network. Initially, "IN" is defined to have the meaning "Internet", but other values MAY be registered in the future.

The second sub-field ("<addrtype>") is the address type. This allows SDP to be used for sessions that are not IP based. This memo only defines IP4 and IP6, but other values MAY be registered in the future.

The third sub-field ("<connection-address>") is the connection address. OPTIONAL sub-fields MAY be added after the connection address depending on the value of the <addrtype> field.

Sessions using an IPv4 multicast connection address MUST also have a time to live (TTL) value present in addition to the multicast address. The TTL and the address together define the scope with which multicast packets sent in this conference will be sent. TTL values MUST be in the range 0-255. Although the TTL MUST be specified, its use to scope multicast traffic is deprecated; applications SHOULD use an administratively scoped address instead.

The TTL for the session is appended to the address using a slash as a separator. An example is:

c=IN IP4 224.2.36.42/127

> Multiple addresses or "c=" lines MAY be specified on a per-media basis only if they provide multicast addresses for different layers in a hierarchical or layered encoding scheme. They MUST NOT be specified for a session-level "c=" field. The slash notation for multiple addresses described above MUST NOT be used for IP unicast addresses.

The `connection` class derives from `attrs` and is used to represent the connection data described before. The individual properties are `nettype` (str), `addrtype` (str), `address` (str) and optionally `ttl` (int) and `count` (int). The constructor takes an optional string `value`. If the value is supplied, it is parsed into the individual properties. Alternatively, the application can construct the object by supplying the individual properties as attribute-value pairs.

```
class connection(attrs):
    def __init__(self, value=None, **kwargs):
        if value:
            self.nettype, self.addrtype, rest = value.split(' ')
            rest = rest.split('/')
            if len(rest) == 1: self.address = rest[0]
            elif len(rest) == 2: self.address, self.ttl = rest[0], int(rest[1])
            else: self.address, self.ttl, self.count = rest[0], int(rest[1]), int(rest[2])
        elif 'address' in kwargs:
            self.address = kwargs.get('address')
            self.nettype = kwargs.get('nettype', 'IN')
            self.addrtype = kwargs.get('addrtype', 'IP4')
            if 'ttl' in kwargs: self.ttl = int(kwargs.get('ttl'))
            if 'count' in kwargs: self.count = int(kwargs.get('count'))
```

As mentioned, the `connection` object can be created in two ways as shown below. The first option parses the value, whereas the second option takes the value of the individual properties. Certain properties have default value when created using the second option, e.g., `addrtype` is “IP4” and `nettype` is “IN”.

```
>>> c = connection(value="c=IN IP4 224.2.1.1/127")
>>> c = connection(address="224.2.1.1", ttl=127)
```

To format a `connection` object into string, we put the properties separated by spaces or other separator we needed in the following method.

```
    def __repr__(self):
        return self.nettype + ' ' + self.addrtype + ' ' + self.address + ('/' + str(self.ttl) if self.ttl else '') + ('/' + str(self.count) if
self.count else '')
```

Media descriptions

From RFC4566 p.22 – Media Descriptions ("m=")

m=<media> <port> <proto> <fmt> ...

A session description may contain a number of media descriptions. Each media description starts with an "m=" field and is terminated by either the next "m=" field or by the end of the session description. A media field has several sub-fields:

<media> is the media type. Currently defined media are "audio", "video", "text", "application", and "message", although this list may be extended in the future.

<port> is the transport port to which the media stream is sent. The meaning of the transport port depends on the network being used as specified in the relevant "c=" field, and on the transport protocol defined in the <proto> sub-field of the media field. Other ports used by the media application (such as the RTP Control Protocol (RTCP) port [19]) MAY be derived algorithmically from the base media port or MAY be specified in a separate attribute (for example, "a=rtcp:").

If non-contiguous ports are used or if they don't follow the parity rule of even RTP ports and odd RTCP ports, the "a=rtcp:" attribute MUST be used. Applications that are requested to send media to a <port> that is odd and where the "a=rtcp:" is present MUST NOT subtract 1 from the RTP port: that is, they MUST send the RTP to the port indicated in <port> and send the RTCP to the port indicated in the "a=rtcp" attribute.

<proto> is the transport protocol. The meaning of the transport protocol is dependent on the address type field in the relevant "c=" field. Thus a "c=" field of IP4 indicates that the transport protocol runs over IP4.

RTP/AVP: denotes RTP used under the RTP Profile for Audio and Video Conferences with Minimal Control running over UDP.

The main reason to specify the transport protocol in addition to the media format is that the same standard media

<fmt> is a media format description. The fourth and any subsequent sub-fields describe the format of the media. The interpretation of the media format depends on the value of the <proto> sub-field.

If the <proto> sub-field is "RTP/AVP" or "RTP/SAVP" the <fmt> sub-fields contain RTP payload type numbers. When a list of payload type numbers is given, this implies that all of these payload formats MAY be used in the session, but the first of these formats SHOULD be used as the default format for the session. For dynamic payload type assignments the "a=rtpmap:" attribute SHOULD be used to map from an RTP payload type number to a media encoding name that identifies the payload format. The "a=fmtp:" attribute MAY be used to specify format parameters.

The `media` class derived from `attrs` class is used to represent the media description line and all the subsequent SDP lines that are attached to this media description line. The properties such as `media` (str), `port` (int), `proto` (str) and `fmt` (list) are defined as described above. The constructor, similar to the `connection` object, takes an optional `value` string. If the `value` is supplied, it gets parsed into individual properties, otherwise the named parameters in the argument list is used to populate the individual properties.

```
class media(attrs):
   def __init__(self, value=None, **kwargs):
      if value:
         self.media, self.port, self.proto, rest = value.split(' ', 3)
         self.port = int(self.port)
         self.fmt = []
         for f in rest.split(' '):
            a = attrs(); a.pt = f; self.fmt.append(a)
      elif 'media' in kwargs:
         self.media = kwargs.get('media')
         self.port  = int(kwargs.get('port', 0))
         self.proto = kwargs.get('proto', 'RTP/AVP')
         self.fmt   = kwargs.get('fmt', [])
```

There are two ways to create a `media` object as shown below. In the first option the supplied `value` string is parsed, and in the second option the parameters populate the properties of the object. In the second option certain parameters take the default values, e.g., default values for `port` and `proto` are 0 and 'RTP/AVP' respectively.

```
>>> m = media(value="audio 8000 RTP/AVP 0 3 8")
>>> m = media(media="audio", port=8000, fmt=[attrs(pt=0), attrs(pt=3), attrs(pt=8)])
```

Since the `media` object also stores the media description specific attributes, the formatting is slightly more complicated to generate multiple SDP lines. Secondly, the format description attributes are stored differently than the other attributes.

To format a `media` object we first print the media description ("m=") SDP line using the `media`, `port`, `proto` and `fmt` properties. Only the payload type (`pt`) property is used from individual elements in the `fmt` format list.

```
    def __repr__(self):
        result = self.media + ' ' + str(self.port) + ' ' + self.proto + ' ' + ' '.join(map(lambda x: str(x.pt), self.fmt))
```

Then it prints out the additional headers such as "i=", "c=", "b=", "k=" and various "a=" SDP lines that are associated with this media description object. If the header is a multiple instance header then it can occur multiple times, and the value is assumed to be a list.

```
        for k in filter(lambda x: x in self, 'icbka'): # order is important
            if k not in SDP._multiple: # single header
                result += '\r\n' + k + '=' + str(self[k])
            else:
                for v in self[k]:
                    result += '\r\n' + k + '=' + str(v)
```

Finally, the "a=rtpmap:" attributes are appended from the `fmt` format list. Because of the ordering restrictions on the headers, this should appear at the end. The formatted string is then returned as the formatted media description which contains the value of the "m=" line followed by name and value of all the other SDP lines that are associated with this "m=" line. Note that the header name, "m", and equals character, "=", are not present in the returned string representing the value of this `media` object.

```
        for f in self.fmt:
            if f.name:
                result += '\r\n' + 'a=rtpmap:' + str(f.pt) + ' ' + f.name + '/' + str(f.rate) + (f.params and ('/'+f.params) or '')
        return result
```

Parsing

Now that we have defined the basic components, let's define the internal parsing routine for the SDP class.

From RFC4566 p.8 – An SDP session description consists of a number of lines of text of the form:

<type>=<value>

where <type> MUST be exactly one case-significant character and <value> is structured text whose format depends on <type>. In general, <value> is either a number of fields delimited by a single space character or a free format string, and is case-significant unless a specific field defines otherwise. Whitespace MUST NOT be used on either side of the "=" sign.

An SDP session description consists of a session-level section followed by zero or more media-level sections. The session-level part starts with a "v=" line and continues to the first media-level section. Each media-level section starts with an "m=" line and continues to the next media-level section or end of the whole session description. In general, session-level values are the default for all media unless overridden by an equivalent media-level value.

The connection ("c=") and attribute ("a=") information in the session-level section applies to all the media of that session unless overridden by connection information or an attribute of the same name in the media description.

The following method takes the `text` string to parse into this SDP object. First we split the string into individual lines. Care must be taken in treating "\n" as same as "\r\n" for interoperability with implementations that generate "\n" as line termination instead of "\r\n". Since various attributes can be either global session attribute or media specific attribute, depending on whether they appear before any "m=" line or after, we need to keep a state variable, g, to indicate whether we are parsing the global session context or the local media description context.

```
def _parse(self, text):
    g = True # whether we are in global line or per media line?
    for line in text.replace('\r\n', '\n').split('\n'):
        … # per line parsing
```

Each line is then split into the header name and value. Note that instead of using the `split` method we use the `partition` method, because the partition needs to be done only once across the given token "=" instead of tokenizing the string using the `split` method. The `strtok` and `strtok_r` functions in the C programming language are equivalent to the `split` method of Python, and should be used with care.

```
        k, sep, v = line.partition('=')
```

If the header name is recognized to be implemented by the specific classes we defined earlier, then we create those specific objects such as `originator`, `connection` and `media`, to parse the header value.

```
        if k == 'o': v = SDP.originator(v)
        elif k == 'c': v = SDP.connection(v)
        elif k == 'm': v = SDP.media(v)
```

Since there can be multiple instances of the "m=" line in the SDP data, the property `m` is defined as a list. Each element in the list is of type `media` object. Since the attributes can be either in the global session context or in the local media description context, we also identify the context for an attribute. In particular, if property `m` doesn't exist then we are in the global context, otherwise we are in the media context.

```
        if k == 'm':   # new m= line
            if not self['m']:
```

```
            self['m'] = []
        self['m'].append(v)
        obj = self['m'][-1]
    elif self['m']:  # not in global
        obj = self['m'][-1]
        ... # store the attribute or other media specific header
    else:        # global header
        obj = self
        ... # store the attribute in global context
```

At this point the `obj` variable points to the appropriate context, either the global `SDP` object or the local `media` object, to which the new header needs to be added.

Adding the new header in the global context is straight forward – if the header is multiple instance header then create a list and append the value to the list, otherwise set the value of the header name property in the `SDP` object. When accessing the property, a multiple-instance header returns a `list` of string values whereas a single instance header returns the string value, e.g., `SDP.a` is a list whereas `SDP.s` is a single string value.

```
obj[k] = ((k in SDP._multiple) and ((k in obj) and (obj[k]+v) or [v])) or v
```

Adding a new header line in the media context is also similar, with one exception. If the header represents a "a=rtpmap:" line, then that needs to be parsed into the format `fmt` list of the `media` object.

> From RFC4566 p.25 – a=rtpmap:<payload type> <encoding name>/<clock rate> [/<encoding parameters>]
>
> This attribute maps from an RTP payload type number (as used in an "m=" line) to an encoding name denoting the payload format to be used. It also provides information on the clock rate and encoding parameters. It is a media-level attribute that is not dependent on charset.
>
> Although an RTP profile may make static assignments of payload type numbers to payload formats, it is more common for that assignment to be done dynamically using "a=rtpmap:" attributes. As an example of a static payload type, consider u-law PCM coded single-channel audio sampled at 8 kHz. This is completely defined in the RTP Audio/Video profile as payload type 0, so there is no need for an "a=rtpmap:" attribute, and the media for such a stream sent to UDP port 49232 can be specified as:

```
m=audio 49232 RTP/AVP 0
```

> An example of a dynamic payload type is 16-bit linear encoded stereo audio sampled at 16 kHz. If we wish to use the dynamic RTP/AVP payload type 98 for this stream, additional information is required to decode it:
>
> Up to one rtpmap attribute can be defined for each media format specified. Thus, we might have the following:

```
m=audio 49230 RTP/AVP 96 97 98
a=rtpmap:96 L8/8000
a=rtpmap:97 L16/8000
a=rtpmap:98 L16/11025/2
```

> RTP profiles that specify the use of dynamic payload types MUST define the set of valid encoding names and/or a means to register encoding names if that profile is to be used with SDP.

For audio streams, <encoding parameters> indicates the number of audio channels. This parameter is OPTIONAL and may be omitted if the number of channels is one, provided that no additional parameters are needed.

For video streams, no encoding parameters are currently specified.

```
            if k == 'a' and v.startswith('rtpmap:'):
                pt, rest = v[7:].split(' ', 1)
                name, sep, rest = rest.partition('/')
                rate, sep, params = rest.partition('/')
                for f in filter(lambda x: x.pt == pt, obj.fmt):
                    f.name = name; f.rate = int(rate); f.params = params or None
            else:
                obj[k] = (k in SDP._multiple and ((k in obj) and (obj[k]+v) or [v])) or v
```

Formatting

Formatting a SDP data is relatively easy. The order of the headers are important. A multiple-instance header is stored as a list and may return in multiple SDP lines. The method to format an SDP is written below.

```
def __repr__(self):
    result = ''
    for k in filter(lambda x: x in self, 'vosiuepcbtam'): # order is important
        if k not in SDP._multiple: # single header
            result += k + '=' + str(self[k]) + '\r\n'
        else:
            for v in self[k]:
                result += k + '=' + str(v) + '\r\n'
    return result
```

Testing

Once we have finished the implementation of the SDP class, we can test the parsing and formatting function as follows:

```
>>> s = '''v=0\r
o=jdoe 2890844526 2890842807 IN IP4 10.47.16.5\r
s=SDP Seminar\r
i=A Seminar on the session description protocol\r
u=http://www.example.com/seminars/sdp.pdf\r
e=j.doe@example.com (Jane Doe)\r
c=IN IP4 224.2.17.12/127\r
t=2873397496 2873404696\r
a=recvonly\r
m=audio 49170 RTP/AVP 0\r
m=video 51372 RTP/AVP 99\r
a=rtpmap:99 h263-1998/90000\r
```

```
'''
>>>  sdp = SDP(s)
>>> print str(sdp) == s
True
>>> print sdp.m[0].port
49170
>>> print sdp.m[1].fmt[0]
{ pt: 99, name: "h263-1998", rate: 90000, params: "" }
```

Now that we have described the implementation of SDP, let's move on to using it in SIP telephony. As mentioned before RFC3264 defines the offer-answer model which is used in the SIP session negotiation between two parties.

Offer-answer in SIP

> From RFC3264 p.1 – This document defines a mechanism by which two entities can make use of the Session Description Protocol (SDP) to arrive at a common view of a multimedia session between them. In the model, one participant offers the other a description of the desired session from their perspective, and the other participant answers with the desired session from their perspective. This offer/answer model is most useful in unicast sessions where information from both participants is needed for the complete view of the session. The offer/answer model is used by protocols like the Session Initiation Protocol (SIP).
>
> The means by which the offers and answers are conveyed are outside the scope of this document. The offer/answer model defined here is the mandatory baseline mechanism used by the Session Initiation Protocol (SIP).

We implement the offer-answer model in our module named `rfc3264`.

Usage

Before implementing the module, let's list down the expected behavior of the module. The module should define two methods: `createOffer` and `createAnswer`, to create session description for an offer or answer respectively. We reuse the SDP and media definitions from the previous module `rfc4566`.

```
>>> from rfc4566 import SDP, attrs as format
```

Media can be described using the `media` object. The following code defines two media objects, one for audio and other for video. The audio has two formats: PCMU and PCMA whereas video has one format H.261.

```
>>> audio = SDP.media(media='audio', port='9000')
>>> audio.fmt = [format(pt=0, name='PCMU', rate=8000), format(pt=8, name='PCMA', rate=8000)]
>>> video = SDP.media(media='video', port='9002')
>>> video.fmt = [format(pt=31, name='H261', rate=90000)]
```

Now the application can create a new offer using these media description as follows.

```
>>> offer = createOffer([audio, video])
```

To test if the `offer` contains a valid `SDP` object, you can print the `offer`.

```
>>> print str(offer)
```

When the offer is received by the answerer, it can use the following code to generate the answer SDP. Support that the answerer wants to support PCMU and GSM audio but no video.

```
>>> audio = SDP.media(media='audio', port='8020')
>>> audio.fmt = [format(pt=0), format(pt=3)]  # for known payload types, description is optional
>>> answer = createAnswer([audio], offer)
```

Now suppose that the offerer wants to change the offer, e.g., using SIP re-INVITE, by removing video from the offer, it should reuse the previous offer as follows.

```
>>> newOffer = createOffer([audio], offer)
```

Thus, the offer can be created either from empty state or from a previous offer, whereas an answer is always created from a previous offer.

Generating the offer

To start the implementation, please note that we need to use the definitions from the `rfc4566` module. Although RFC 3264 uses old specification of SDP as in RFC 2327, we use the new specification of SDP as in RFC 4566.

```
from std.rfc4566 import SDP, attrs as format
import socket
```

We also define a module level flag to enable or disable the trace which helps us in debugging the module. The default is to disable the trace, but a programme may enable it by setting it to `True`.

```
_debug = False
```

> From RFC3264 p.4 – Media Stream: From RTSP [8], a media stream is a single media instance, e.g., an audio stream or a video stream as well as a single whiteboard or shared application group. In SDP, a media stream is described by an "m=" line and its associated attributes.

We use the `media` class defined in `SDP` to represent the media stream.

The offer (and answer) MUST be a valid SDP message, as defined by RFC 2327, with one exception. RFC 2327 mandates that either an e or a p line is present in the SDP message. This specification relaxes that constraint; an SDP formulated for an offer/answer application MAY omit both the e and p lines. The numeric value of the session id and version in the o line MUST be representable with a 64 bit signed integer. The initial value of the version MUST be less than (2**62)-1, to avoid rollovers. Although the SDP specification allows for multiple session descriptions to be concatenated together into a large SDP message, an SDP message used in the offer/answer model MUST contain exactly one session description.

The SDP "s=" line conveys the subject of the session, which is reasonably defined for multicast, but ill defined for unicast. For unicast sessions, it is RECOMMENDED that it consist of a single space character (0x20) or a dash (-).

Unfortunately, SDP does not allow the "s=" line to be empty.

The SDP "t=" line conveys the time of the session. Generally, streams for unicast sessions are created and destroyed through external signaling means, such as SIP. In that case, the "t=" line SHOULD have a value of "0 0".

The offer will contain zero or more media streams (each media stream is described by an "m=" line and its associated attributes). Zero media streams implies that the offerer wishes to communicate, but that the streams for the session will be added at a later time through a modified offer. The streams MAY be for a mix of unicast and multicast; the latter obviously implies a multicast address in the relevant "c=" line(s).

Construction of each offered stream depends on whether the stream is multicast or unicast.

```
def createOffer(streams, previous=None, **kwargs):
   '''Create an offer SDP using local (streams) list of media Stream objects.
   If a previous offer/answer SDP is specified then it creates a modified offer.
   Additionally, the optional keyword arguments such as e and p can be specified.'''
   s = SDP()
   s.v = '0'
   for a in "iep": # add optioanl e and p headers if present
      if a in kwargs: s[a] = kwargs[a]
   s.o = SDP.originator(previous and str(previous.o) or None)
   if previous: s.o.version = s.o.version + 1
   s.s = '-'
   s.t = ['0 0'] # because t= can appear multiple times, it is a list.
   s.m = streams
   return s
```

Generating the answer

We simplify our implementation to support only the unicast addresses, and not worry about various headers. The following implementation just matches the media description lines and the format list correctly from the offer and the supplied locally supported media streams.

```
def createAnswer(streams, offer, **kwargs):
   '''Create an answer SDP for the remote offer SDP using local (streams) list of  media Stream objects.'''
   s = SDP()
   s.v = '0'
   for a in "iep":
      if a in kwargs: s[a] = kwargs[a]
```

From RFC3264 p.9 – The answer to an offered session description is based on the offered session description. If the answer is different from the offer in any way (different IP addresses, ports, etc.), the origin line MUST be different in the answer, since the answer is generated by a different entity. In that case, the version number in the "o=" line of the answer is unrelated to the version number in the o line of the offer.

```
s.o = SDP.originator()
s.s = '-'
```

The "t=" line in the answer MUST equal that of the offer. The time of the session cannot be negotiated.

```
s.t = offer.t
s.m = []
streams = list(streams) # so that original stream is not modified
```

For each "m=" line in the offer, there MUST be a corresponding "m=" line in the answer. The answer MUST contain exactly the same number of "m=" lines as the offer. This allows for streams to be matched up based on their order. This implies that if the offer contained zero "m=" lines, the answer MUST contain zero "m=" lines.

```
for your in offer.m: # for each m= line in offer
   my   = None      # answered stream
   for i in range(0, len(streams)):
```

If a stream is offered with a unicast address, the answer for that stream MUST contain a unicast address. The media type of the stream in the answer MUST match that of the offer.

If a stream is offered as sendonly, the corresponding stream MUST be marked as recvonly or inactive in the answer. If a media stream is listed as recvonly in the offer, the answer MUST be marked as sendonly or inactive in the answer. If an offered media stream is listed as sendrecv (or if there is no direction attribute at the media or session level, in which case the stream is sendrecv by default), the corresponding stream in the answer MAY be marked as sendonly, recvonly, sendrecv, or inactive. If an offered media stream is listed as inactive, it MUST be marked as inactive in the answer.

For streams marked as recvonly in the answer, the "m=" line MUST contain at least one media format the answerer is willing to receive with from amongst those listed in the offer. The stream MAY indicate additional media formats, not listed in the corresponding stream in the offer, that the answerer is willing to receive. For streams marked as sendonly in the answer, the "m=" line MUST contain at least one media format the answerer is willing to send from amongst those listed in the offer. For streams marked as sendrecv in the answer, the "m=" line MUST contain at least one codec the answerer is willing to both send and receive, from amongst those listed in the offer. The stream MAY indicate additional media formats, not listed in the corresponding stream in the offer, that the answerer is willing to send or receive (of course, it will not be able to send them at this time, since it was not listed in the offer). For streams marked as inactive in the answer, the list of media formats is constructed based on the offer. If the offer was sendonly, the list is constructed as if the answer were recvonly. Similarly, if the offer was recvonly, the list is constructed as if the answer were sendonly, and if the offer was sendrecv, the list is constructed as if the answer were sendrecv. If the offer was inactive, the list is constructed as if the offer were actually sendrecv and the answer were sendrecv.

The connection address and port in the answer indicate the address where the answerer wishes to receive media (in the case of RTP, RTCP will be received on the port which is one higher unless there is an explicit indication otherwise). This address and port MUST be present even for sendonly streams; in the case of RTP, the port one higher is still used to receive RTCP.

```
if streams[i].media == your.media: # match the first stream in streams
   my = SDP.media(str(streams[i])) # found, hence
   del streams[i]  #  remove from streams so that we don't match again for another m=
   found = []
```

In the case of RTP, if a particular codec was referenced with a specific payload type number in the offer, that same payload type number SHOULD be used for that codec in the answer. Even if the same payload type number is used, the answer MUST contain rtpmap attributes to define the payload type mappings for dynamic payload types, and SHOULD contain mappings for static payload types. The media formats in the "m=" line MUST be listed in order of preference, with the first format listed being preferred. In this case, preferred means that the offerer SHOULD use the format with the highest preference from the answer.

Although the answerer MAY list the formats in their desired order of preference, it is RECOMMENDED that unless there is a specific reason, the answerer list formats in the same relative order they were present in the offer. In other words, if a stream in the offer lists audio codecs 8, 22 and 48, in that order, and the answerer only supports codecs 8 and 48, it is RECOMMENDED that, if the answerer has no reason to change it, the ordering of codecs in the answer be 8, 48, and not 48, 8. This helps assure that the same codec is used in both directions.

The interpretation of fmtp parameters in an offer depends on the parameters. In many cases, those parameters describe specific configurations of the media format, and should therefore be processed as the media format value itself would be. This means that the same fmtp parameters with the same values MUST be present in the answer if the media format they describe is present in the answer. Other fmtp parameters are more like parameters, for which it is perfectly acceptable for each agent to use different values. In that case, the answer MAY contain fmtp parameters, and those MAY have the same values as those in the offer, or they MAY be different. SDP extensions that define new parameters SHOULD specify the proper interpretation in offer/answer.

The answerer MAY include a non-zero ptime attribute for any media stream; this indicates the packetization interval that the answerer would like to receive. There is no requirement that the packetization interval be the same in each direction for a particular stream.

The answerer MAY include a bandwidth attribute for any media stream; this indicates the bandwidth that the answerer would like the offerer to use when sending media. The value of zero is allowed, interpreted as described in Section 5.

```
for fy in your.fmt:  # all offered formats, find the matching pairs
   for fm in my.fmt:# the preference order is from offer, hence do for fy, then for fm.
      try: fmpt, fypt = int(fm.pt), int(fy.pt) # try using numeric payload type
      except: fmpt = fypt = -1
      if 0<=fmpt<32 and 0<=fypt<32 and fmpt == fypt \
      or fmpt<0 and fypt<0 and fm.pt == fy.pt \
      or fm.name == fy.name and fm.rate == fy.rate and fm.count == fy.count: # we don't match the params
         found.append((fy, fm)); break
```

If the answerer has no media formats in common for a particular offered stream, the answerer MUST reject that media stream by setting the port to zero.

```
if found: # we found some matching formats, put them in
   my.fmt = map(lambda x: x[0], found) # use remote's fy including fy.pt
else:
   my.fmt = [format(pt=0)] # no match in formats, but matched media, must put a format with payload type 0
   my.port = 0          #   and reset the port.
```

An offered stream MAY be rejected in the answer, for any reason. If a stream is rejected, the offerer and answerer MUST NOT generate media (or RTCP packets) for that stream. To reject an offered stream, the port number in the corresponding stream in the answer MUST be set to zero. Any media formats listed are ignored. At least one MUST be present, as specified by SDP.

```
    if not my: # did not match the stream, must put a stream with port = 0
        my = SDP.media(str(your))
        my.port = 0
    s.m.append(my) # append it to our media
```

If there are no media formats in common for all streams, the entire offered session is rejected.

```
valid = False
for my in s.m: # check if any valid matching stream is present with valid port
    if my.port != 0:
        valid = True
        break

return valid and s or None  # if no valid matching stream found, return None
```

Once the answerer has sent the answer, it MUST be prepared to receive media for any recvonly streams described by that answer. It MUST be prepared to send and receive media for any sendrecv streams in the answer, and it MAY send media immediately. The answerer MUST be prepared to receive media for recvonly or sendrecv streams using any media formats listed for those streams in the answer, and it MAY send media immediately. When sending media, it SHOULD use a packetization interval equal to the value of the ptime attribute in the offer, if any was present. It SHOULD send media using a bandwidth no higher than the value of the bandwidth attribute in the offer, if any was present. The answerer MUST send using a media format in the offer that is also listed in the answer, and SHOULD send using the most preferred media format in the offer that is also listed in the answer. In the case of RTP, it MUST use the payload type numbers from the offer, even if they differ from those in the answer.

In this chapter we have implemented the session description protocol and offer-answer model that are needed for SIP telephony. Next we describe the basic and digest authentication.

Authentication

Implementing RFC 2617 for Basic and Digest authentication

SIP uses the authentication mechanism defined for HTTP. In particular, the digest authentication defined in RFC2617 provides a challenge-response authentication that does not send the password in clear text.

We implement the authentication module named `rfc2617`. We would like to support both “Basic” and “Digest” authentication method defined in RFC2617. Although SIP does not allow “Basic” authentication because it sends the password in clear, we do implement the mechanism as it can work well with underlying transport security between the client and the server.

Usage

Before we implement the module, let’s discuss the expected usage of the module. When a client (UAC) sends a SIP request to the server (UAS), the server may challenge the request by responding with a 401 or 407 response. The server puts the `WWW-Authenticate` or `Proxy-Authenticate` header in the response. Let’s assume that the server invokes the `createAuthenticate` method to create the header value.

```
>>> print createAuthenticate('Basic', realm='iptel.org')
Basic realm="iptel.org"
>>> print createAuthenticate('Digest', realm='iptel.org', domain='sip:iptel.org', nonce='somenonce')
Digest realm="iptel.org", domain="sip:iptel.org", qop="auth", nonce="somenonce", opaque="", stale=FALSE,
algorithm=MD5
```

When the client wants to re-send the request with the authorization credentials, it puts the `Authorization` or `Proxy-Authorization` header in the new request which supplies the credentials. It invokes the `createAuthorization` method to create the header value.

```
>>> print createAuthorization('Basic realm="WallyWorld"', 'Aladdin', 'open sesame')
Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
>>> context = {'cnonce':'0a4f113b', 'nc': 0}
>>> print createAuthorization('Digest realm="testrealm@host.com", qop="auth",
nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093", opaque="5ccc069c403ebaf9f0171e9517f40e41"', 'Mufasa', 'Circle Of
Life', '/dir/index.html', 'GET', None, context)
Digest cnonce="0a4f113b",nc=00000001,nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
opaque="5ccc069c403ebaf9f0171e9517f40e41",qop=auth,realm="testrealm@host.com",response="6629fae49393a053
97450978507c4ef1",uri="/dir/index.html",username="Mufasa"
```

Let’s now focus on implementing these two public methods in our module.

Quoting a string

> From RFC2617 p.3 – HTTP provides a simple challenge-response authentication mechanism that MAY be used by a server to challenge a client request and by a client to provide authentication information. It uses an extensible, case-insensitive token to identify the authentication scheme, followed by a comma-separated list of attribute-value pairs which carry the parameters necessary for achieving authentication via that scheme.
>
> ```
> auth-scheme    = token
> auth-param     = token "=" ( token | quoted-string )
> ```

Let's define the `quote` and `unquote` internal methods that can quote or unquote a string if needed.

```
_quote   = lambda s: '"' + s + '"' if not s or s[0] != '"' != s[-1] else s
_unquote = lambda s: s[1:-1] if s and s[0] == '"' == s[-1] else s
```

Create Authenticate

The method takes the `authMethod` argument which is either "Basic" or "Digest" (case-insensitive), followed by bunch of named parameters. Possible parameter names are `realm`, `domain`, `qop`, `nonce`, `opaque`, `stale` and `algorithm`. Usually the `realm` is mandatory for "Basic" authentication, and `realm` and `domain` for "Digest". Other parameters if needed but not specified, take the default values.

```
from random import randint
from hashlib import md5
from base64 import b64encode
import time
…
def createAuthenticate(authMethod='Digest', **kwargs):
```

The "Basic" authentication's header value is straightforward which just puts the `realm` as quoted string in the authentication parameters.

```
    if authMethod.lower() == 'basic':
        return 'Basic realm=%s'%(_quote(kwargs.get('realm', '')))
```

The "Digest" authentication creates the list of authentication parameters from the supplied values or the defaults, such that the parameters are put in order specified below. I have seen some implementation that doesn't interoperate if the order of the parameters is not same as what is presented in the specification. Only the `stale` and `algorithm` parameters are unquoted, others are quoted strings.

```
    elif authMethod.lower() == 'digest':
        predef = ('realm', 'domain', 'qop', 'nonce', 'opaque', 'stale', 'algorithm')
        unquoted = ('stale', 'algorithm')
        now = time.time();  nonce = b64encode('%d %s'%(now, md5('%d:%d'%(now, id(createAuthenticate)))))
        nonce = kwargs.get('nonce', nonce)
        default = dict(realm='', domain='', opaque='', stale='FALSE', algorithm='MD5', qop='auth', nonce=nonce)
        # put predef attributes in order before non predef attributes
```

```
    kv = map(lambda x: (x, kwargs.get(x, default[x])), predef) + filter(lambda x: x[0] not in predef, kwargs.items())
    return 'Digest ' + ', '.join(map(lambda y: '%s=%s'%(y[0], _quote(y[1]) if y[0] not in unquoted else y[1]), kv))
```

The method gives an error if the `authMethod` is unsupported.

```
else: raise ValueError, 'invalid authMethod%s'%(authMethod)
```

Create Authorization

From RFC2617 p.3 – The 401 (Unauthorized) response message is used by an origin server to challenge the authorization of a user agent. This response MUST include a WWW-Authenticate header field containing at least one challenge applicable to the requested resource. The 407 (Proxy Authentication Required) response message is used by a proxy to challenge the authorization of a client and MUST include a Proxy-Authenticate header field containing at least one challenge applicable to the proxy for the requested resource.

```
challenge   = auth-scheme 1*SP 1#auth-param
```

A user agent that wishes to authenticate itself with an origin server--usually, but not necessarily, after receiving a 401 (Unauthorized)--MAY do so by including an Authorization header field with the request. A client that wishes to authenticate itself with a proxy--usually, but not necessarily, after receiving a 407 (Proxy Authentication Required)--MAY do so by including a Proxy-Authorization header field with the request. Both the Authorization field value and the Proxy-Authorization field value consist of credentials containing the authentication information of the client for the realm of the resource being requested. The user agent MUST choose to use one of the challenges with the strongest auth-scheme it understands and request credentials from the user based upon that challenge.

```
credentials = auth-scheme #auth-param
```

The following method builds the `Authorization` header value for the specified `challenge`. The `challenge` argument must be a string representing the `WWW-Authenticate` (or `Proxy-Authenticate`) header value. The method parses it to identify the various authentication parameters. The other arguments are as follows: the `username` and `password` parameters supply the credentials for authentication, the `uri`, `method` and `entityBody` parameters supply those properties of the request which are used in building the digest credentials, and finally the `context` argument is used to store the state for digest authorization, such as `cnonce` and `nonceCount`, if available.

```
def createAuthorization(challenge, username, password, uri=None, method=None, entityBody=None, context=None):
    authMethod, sep, rest = challenge.strip().partition(' ')
    ch, cr = dict(), dict() # challenge and credentials
    cr['password']   = password
    cr['username']   = username
```

From RFC2617 p.5 – The "basic" authentication scheme is based on the model that the client must authenticate itself with a user-ID and a password for each realm. The realm value should be considered an opaque string which can only be compared for equality with other realms on that server. The server will service the request only if it can validate the user-ID and password for the protection space of the Request-URI. There are no optional authentication parameters. For Basic, the framework above is utilized as follows:

```
challenge   = "Basic" realm
```

```
credentials = "Basic" basic-credentials
```

Upon receipt of an unauthorized request for a URI within the protection space, the origin server MAY respond with a challenge like the following:

```
WWW-Authenticate: Basic realm="WallyWorld"
```

where "WallyWorld" is the string assigned by the server to identify the protection space of the Request-URI. A proxy may respond with the same challenge using the Proxy-Authenticate header field.

We delegate this function into the `basic` method defined later.

```
if authMethod.lower() == 'basic':
   return authMethod + ' ' + basic(cr)
```

From RFC2617 p.6 – Like Basic Access Authentication, the Digest scheme is based on a simple challenge-response paradigm. The Digest scheme challenges using a nonce value. A valid response contains a checksum (by default, the MD5 checksum) of the username, the password, the given nonce value, the HTTP method, and the requested URI. In this way, the password is never sent in the clear. Just as with the Basic scheme, the username and password must be prearranged in some fashion not addressed by this document.

```
elif authMethod.lower() == 'digest':
   for n,v in map(lambda x: x.strip().split('='), rest.split(',') if rest else []):
      ch[n.lower().strip()] = _unquote(v.strip())
   # TODO: doesn't work if embedded ',' in value, e.g., qop="auth,auth-int"
```

If a server receives a request for an access-protected object, and an acceptable Authorization header is not sent, the server responds with a "401 Unauthorized" status code, and a WWW-Authenticate header as per the framework defined above, which for the digest scheme is utilized as follows:

```
challenge     =  "Digest" digest-challenge
digest-challenge = 1#( realm | [ domain ] | nonce |
                       [ opaque ] |[ stale ] | [ algorithm ] |
                       [ qop-options ] | [auth-param] )
domain        = "domain" "=" <"> URI ( 1*SP URI ) <">
URI           = absoluteURI | abs_path
nonce         = "nonce" "=" nonce-value
nonce-value   = quoted-string
opaque        = "opaque" "=" quoted-string
stale         = "stale" "=" ( "true" | "false" )
algorithm     = "algorithm" "=" ( "MD5" | "MD5-sess" |  token )
qop-options   = "qop" "=" <"> 1#qop-value <">
qop-value     = "auth" | "auth-int" | token
```

```
   for y in filter(lambda x: x in ch, ['username', 'realm', 'nonce', 'opaque', 'algorithm']):
      cr[y] = ch[y]
   cr['uri']       = uri
   cr['httpMethod'] = method
   if 'qop' in ch:
      if context and 'cnonce' in context:
         cnonce, nc = context['cnonce'], context['nc'] + 1
      else:
         cnonce, nc = H(str(randint(0, 2**31))), 1
      if context:
```

```
        context['cnonce'], context['nc'] = cnonce, nc
    cr['qop'], cr['cnonce'], cr['nc'] = 'auth', cnonce, '%08x'% nc
```

The client is expected to retry the request, passing an Authorization header line, which is defined according to the framework above, utilized as follows.

```
credentials      = "Digest" digest-response
digest-response  = 1#( username | realm | nonce |
                   digest-uri  | response | [ algorithm ] |
                   [cnonce] |  [opaque] | [message-qop] |
                   [nonce-count]  | [auth-param] )
username         = "username" "=" username-value
username-value   = quoted-string
digest-uri       = "uri" "=" digest-uri-value
digest-uri-value = request-uri   ; As specified by HTTP/1.1
message-qop      = "qop" "=" qop-value
cnonce           = "cnonce" "=" cnonce-value
cnonce-value     = nonce-value
nonce-count      = "nc" "=" nc-value
nc-value         = 8LHEX
response         = "response" "=" request-digest
```

```
    cr['response'] = digest(cr)
    items = sorted(filter(lambda x: x not in ['name', 'authMethod', 'value', 'httpMethod', 'entityBody', 'password'], cr))
    return authMethod + ' ' + ','.join(map(lambda y: '%s=%s'%(y, (cr[y] if y == 'qop' or y == 'nc' else _quote(cr[y]))), items))
else:
    raise ValueError, 'Invalid auth method -- ' + authMethod
```

In this document the string obtained by applying the digest algorithm to the data "data" with secret "secret" will be denoted by KD(secret, data), and the string obtained by applying the checksum algorithm to the data "data" will be denoted H(data). The notation unq(X) means the value of the quoted-string X without the surrounding quotes.

For the "MD5" and "MD5-sess" algorithms

```
H(data) = MD5(data)
```

and

```
KD(secret, data) = H(concat(secret, ":", data))
```

```
H = lambda d: md5(d).hexdigest()
KD = lambda s, d: H(s + ':' + d)
```

The first time the client requests the document, no Authorization header is sent, so the server responds with:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest
        realm="testrealm@host.com",
        qop="auth,auth-int",
        nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
        opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

The client may prompt the user for the username and password, after which it will respond with a new request, including the following Authorization header:

```
Authorization: Digest username="Mufasa",
```

```
realm="testrealm@host.com",
nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
uri="/dir/index.html",
qop=auth,
nc=00000001,
cnonce="0a4f113b",
response="6629fae49393a05397450978507c4ef1",
opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

```
>>> input = {'httpMethod':'GET', 'username':'Mufasa', 'password': 'Circle Of Life', 'realm':'testrealm@host.com',
'algorithm':'md5', 'nonce':'dcd98b7102dd2f0e8b11d0f600bfb0c093', 'uri':'/dir/index.html', 'qop':'auth', 'nc': '00000001',
'cnonce':'0a4f113b', 'opaque':'5ccc069c403ebaf9f0171e9517f40e41'}
>>> print digest(input)
"6629fae49393a05397450978507c4ef1"
```

Digest

We define the `digest` method to create such a digest response.

```
def digest(cr):
    algorithm, username, realm, password, nonce, cnonce, nc, qop, httpMethod, uri, entityBody \
      = map(lambda x: cr[x] if x in cr else None, ['algorithm', 'username', 'realm', 'password', 'nonce', 'cnonce', 'nc', 'qop',
'httpMethod', 'uri', 'entityBody'])
```

If the "algorithm" directive's value is "MD5" or is unspecified, then A1 is:

```
A1      = unq(username-value) ":" unq(realm-value) ":" passwd
```

where

```
passwd  = < user's password >
```

If the "algorithm" directive's value is "MD5-sess", then A1 is calculated only once - on the first request by the client following receipt of a WWW-Authenticate challenge from the server. It uses the server nonce from that challenge, and the first client nonce value to construct A1 as follows:

```
A1      = H( unq(username-value) ":" unq(realm-value)
             ":" passwd )
             ":" unq(nonce-value) ":" unq(cnonce-value)
```

```
if algorithm and algorithm.lower() == 'md5-sess':
    A1 = H(username + ':' + realm + ':' + password) + ':' + nonce + ':' + cnonce
else:
    A1 = username + ':' + realm + ':' + password
```

If the "qop" directive's value is "auth" or is unspecified, then A2 is:

```
A2       = Method ":" digest-uri-value
```

If the "qop" value is "auth-int", then A2 is:

```
A2       = Method ":" digest-uri-value ":" H(entity-body)
```

```
if not qop or qop == 'auth':
   A2 = httpMethod + ':' + str(uri)
else:
   A2 = httpMethod + ':' + str(uri) + ':' + H(str(entityBody))
```

If the "qop" value is "auth" or "auth-int":

```
request-digest  = <"> < KD ( H(A1),     unq(nonce-value)
                                   ":" nc-value
                                   ":" unq(cnonce-value)
                                   ":" unq(qop-value)
                                   ":" H(A2)
                              ) <">
```

If the "qop" directive is not present (this construction is for compatibility with RFC 2069):

```
request-digest =
           <"> < KD ( H(A1), unq(nonce-value) ":" H(A2) ) > <">
```

```
if qop and (qop == 'auth' or qop == 'auth-int'):
   a = nonce + ':' + str(nc) + ':' + cnonce + ':' + qop + ':' + A2
   return _quote(KD(H(A1), nonce + ':' + str(nc) + ':' + cnonce + ':' + qop + ':' + H(A2)))
else:
   return _quote(KD(H(A1), nonce + ':' + H(A2)))
```

Basic

If the user agent wishes to send the userid "Aladdin" and password "open sesame", it would use the following header field:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

```
>>> print basic({'username':'Aladdin', 'password':'open sesame'})
QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

To receive authorization, the client sends the userid and password, separated by a single colon (":") character, within a base64 encoded string in the credentials.

```
basic-credentials = base64-user-pass
base64-user-pass  = <base64 [4] encoding of user-pass,
                    except not limited to 76 char/line>
user-pass   = userid ":" password
userid      = *<TEXT excluding ":">
password    = *TEXT
```

Userids might be case sensitive.

```
def basic(cr):
   return b64encode(cr['username'] + ':' + cr['password'])
```

The authentication module forms an integral part of any SIP implementation for both client as well as server side. Next we explore the client specific extensions for SIP telephony.

Part 3 Client extensions

this part extends the basic implementation to support various client-side features such as media transport, traversal of network address translator (NAT) and firewall, instant messaging and presence, contact list management and audio-video tools.

Real-time Transport Protocol

Implementing RTP/RTCP as per RFC 3550, RFC 3551

The Real-time Transport Protocol (RTP) defines a standardized packet format for delivering audio and video over the Internet. It is used for several internet protocols such as RTSP for streaming and SIP for multimedia sessions.

> From RFC3550 p.1 – This memorandum describes RTP, the real-time transport protocol. RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services. The data transport is augmented by a control protocol (RTCP) to allow monitoring of the data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality. RTP and RTCP are designed to be independent of the underlying transport and network layers. The protocol supports the use of RTP-level translators and mixers.
>
> Most of the text in this memorandum is identical to RFC 1889 which it obsoletes. There are no changes in the packet formats on the wire, only changes to the rules and algorithms governing how the protocol is used. The biggest change is an enhancement to the scalable timer algorithm for calculating when to send RTCP packets in order to minimize transmission in excess of the intended rate when many participants join a session simultaneously.

The RTP specification inserts a header, typically 12 bytes long, to the audio video payload. This RTP header provides synchronization, timing and sequencing information. RTP works in conjunction with another protocol, namely Real-time Transport Control Protocol (RTCP). RTCP is used to provide various quality feedback and synchronization information. The base specification works for both unicast as well as multicast applications. The implementation of base RTP is straight forward. However, the implementation of RTCP is more involved. Unlike the other standards such as SIP, the specification of RTP and RTCP is presented in the RFCs in very low level details, including source code in C programming language. This helps us for those parts where we can readily port the C source code to Python programming language for our implementation.

In this chapter we will implement RTP and RTCP as per RFC 3550. We will also implement the audio video profile as defined in RFC 3551. Let's create new module named `rfc3550` and `rfc3551` to implement these two specifications.

Implementing RTP and RTCP

At the high level there are four parts in the `rfc3550` module: (1) the `RTP` and `RTCP` classes define the packet format for RTP and RTCP, respectively, including parsing and formatting, (2) the `Session` class defines the control behavior for an RTP session, (3) the `Source` class represents a member or source in a session, and (4) the `Network` class abstracts out the network behavior such as a pair sockets, hence allows us to keep the network transport outside our implementation.

In our module we will use a number of existing libraries such as `struct` for binary packet format, `random` for random number generation, `math` for various math operations, `time` for getting the current time, and `socket` for getting the IP address and performing network transport.

```
import struct, random, math, time, socket
from kutil import getlocaladdr
```

Let's also define a convenience flag to enable or disable the trace in our module.

```
_debug = False
```

Packet format

The packet format for RTP and RTCP follows the binary protocol mechanism. Let's define a convenience function to print some data in binary format, to help us debug out module. Let's assume the `binstr` function converts the supplied string into its binary representation with up to 32 bits per line. The specification also assumes 32-bits boundary for various headers.

```
>>> print binstr('\\x01\\x02\\x03\\x04\\x05\\x06\\x07')
00000001000000100000001100000100
00000101000001100000111-------
```

We implement this function in two steps: first we define a method called `binary` which converts the supplied data into list of strings, where each string is the binary representation of the specific number of consecutive bytes as controlled by the `size` argument. For example, calling `binary(data, size=4)` will return lists containing binary representations of all the 32-bits words in the data.

```
def binary(data, size=4):
    all = ''.join([''.join([('1' if (ord(x) & (1<<(7-y))) else '0') for y in range(0, 8)]) for x in data])
    result, size = [], size*8  # size to bits
    while len(all) >= size:
        result.append(all[:size])
        all = all[size:]
    if len(all)>0:
        result.append(all + '-'*(size-len(all)))
    return result
```

In the second step we define a method to convert this list into a single multi-line string for printing purpose.

```
binstr = lambda x: '\n'.join(binary(x))
```

RTP packet

> From RFC3550 p.8 – RTP packet: A data packet consisting of the fixed RTP header, a possibly empty list of contributing sources (see below), and the payload data. Some underlying protocols may require an encapsulation of the RTP packet to be defined. Typically one packet of the underlying protocol contains a single RTP packet, but several RTP packets MAY be contained if permitted by the encapsulation method.

Let's assume that the RTP class represents an RTP packet. There are two important functions: parsing and formatting. An RTP object can be constructed either from individual elements of the RTP header or from the

received data. In the latter case, it parses the data into the `RTP` object. The formatting function can be implemented using the `__repr__` method to get the string (binary) representation of the object. This allows us to use the object in string context, where it automatically gets the binary formatted value of the packet.

Usage

The following example shows how to construct an RTP packet by specifying the individual header fields using named parameters. The `extn` argument supplies the length as well as the value in a tuple, and the `payload` argument supplies the value in binary form.

```
>>> p1 = RTP(pt=8, seq=12, ts=13, ssrc=14, csrcs=[15, 16], marker=True, extn=(17, '\\x18\\x19\\x1a\\x1b'),
payload='\\x1c\\x1d\\x1e')
```

By printing the hexadecimal representation of the packet, we can confirm that our packet was well formed. In particular, you can check the various headers, the extension field, payload and the final padding byte.

```
>>> print ''.join(['%02x'%ord(x) for x in str(p1)])
b288000c0000000d0000000e0000000f00000010001100011819 1a1b1c1d1e01
```

To further verify the functions, let's construct another RTP packet using the value of the first packet.

```
>>> p2 = RTP(value=str(p1))
```

We can print the individual headers fields of the packet to verify that the values are what were set in the original packet.

```
>>> print p2.pt, p2.seq, p2.ts, p2.ssrc, p2.csrcs, p2.marker, p2.extn, repr(p2.payload)
8 12 13 14 [15, 16] True (17, '\\x18\\x19\\x1a\\x1b') '\\x1c\\x1d\\x1e'
```

The following example demonstrates the binary representation of the RTP packet.

```
>>> print '\\n'.join(binary(str(p2)))
10110010100010000000000000001100
00000000000000000000000000001101
00000000000000000000000000001110
00000000000000000000000000001111
00000000000000000000000000010000
00000000000100010000000000000001
00011000000110010001101000011011
00011100000111010001111000000001
```

Properties

Let's define the `RTP` class. As mentioned above, the constructor takes overloaded set of arguments: either the `value` argument can be supplied containing the binary packet, or the individual header fields can be supplied. These individual header fields are stored as properties in the `RTP` object. The `pt` or payload type is an integer 0-127. The `seq` property is a two-bytes integer representing the sequence number. The `ts` property is a four-bytes integer representing the timestamp. The `ssrc` property is a four-bytes integer representing the synchronization source identifier. The `csrcs` is a list of four-bytes integers, representing the various contributing source identifiers, if any. The `marker` property is a Boolean indicating whether the marker is set or not. The `extn` optional property is a tuple, with first element indicating the length and the second element representing the actual binary data for the extension. The `payload` property represents the actual binary payload data in this packet.

```
class RTP(object):
  def __init__(self, value=None, pt=0, seq=0, ts=0, ssrc=0, csrcs=[], marker=False, extn=None, payload=''):
    if not value: # construct using components
      self.pt, self.seq, self.ts, self.ssrc, self.csrcs, self.marker, self.extn, self.payload = \
      pt, seq, ts, ssrc, csrcs, marker, extn, payload
```

Parsing

If `value` argument is supplied, we parse the `value` into various header field properties as follows. The minimum header size is 12 bytes, otherwise it gives an error. The RTP version number must be 2 otherwise it gives and error.

```
    else: # parse the packet.
      if len(value) < 12: raise ValueError, 'RTP packet must be at least 12 bytes'
      if ord(value[0]) & 0xC0 != 0x80: raise ValueError, 'RTP version must be 2'
```

The first 12 bytes are unpacked into the initial mandatory headers.

```
      px, mpt, self.seq, self.ts, self.ssrc = struct.unpack('!BBHII', value[:12])
      self.marker, self.pt = (mpt & 0x80 and True or False), (mpt & 0x7f)
```

This is followed by an optional list of CSRCs.

```
      self.csrcs, value = ([] if (px & 0x0f == 0) else list(struct.unpack('!'+'I'*(px&0x0f), value[12:12+(px&0x0f)*4]))),
value[12+(px & 0x0f)*4:]
```

If an extension is present it is parsed into the `extn` property.

```
      if px & 0x10:
        xtype, xlen = struct.unpack('!HH', value[:4])
        self.extn, value = (xtype, value[4:4+xlen*4]), value[4+xlen*4:]
      else: self.extn = None
```

Finally, the payload is stored in the `payload` property. Note that if padding is present, the padding bytes are not included in the `payload`.

```
        self.payload = value if px & 0x20 == 0 else value[:len(value)-ord(value[-1])]
```

Formatting

Formatting the RTP object into binary format can be done in a single Python statement as shown below. This example shows the power of the programming language for this kind of implementations.

```
    def __repr__(self):
        return struct.pack('!BBHII', 0x80 | ((len(self.payload)%4 != 0) and 0x20 or 0x00) | (self.extn and 0x10 or 0x00) |
(len(self.csrcs) > 15 and 15 or len(self.csrcs)), \
                    (self.pt & 0x7f) | (self.marker and 1 or 0) << 7, (self.seq & 0xffff), self.ts, self.ssrc) \
                + ''.join(struct.pack('!I', x) for x in self.csrcs[:16]) \
                + ('' if not self.extn else (struct.pack('!HH', self.extn[0] & 0xffff, len(self.extn[1])/4) + self.extn[1])) \
                + self.payload \
                + ('' if (len(self.payload) % 4 == 0) else ('\x00'*(4-len(self.payload)%4-1) + struct.pack('!B', 4-
len(self.payload)%4)))
```

RTCP packet

> RTCP packet: A control packet consisting of a fixed header part similar to that of RTP data packets, followed by structured elements that vary depending upon the RTCP packet type. Typically, multiple RTCP packets are sent together as a compound RTCP packet in a single packet of the underlying protocol; this is enabled by the length field in the fixed header of each RTCP packet.

Usage

Let's assume that the RTCP class implements a compound RTCP packet. For representing an individual packet or a sub-packet we assume the nested class `RTCP.packet`. As with an RTP packet we would like to be able to create an RTCP packet using individual header components. The following example creates a new sender report packet.

```
>>> sr = RTCP.packet(pt=RTCP.SR, ssrc=1, ntp=2, ts=3, pktcount=4, octcount=5, reports=[], extn=None)
```

Similarly, we can create the receiver report with two report elements as follows:

```
>>> r1 = RTCP.packet(ssrc=1, flost=2, clost=3, hseq=4, jitter=5, lsr=6, dlsr=7)
>>> r2 = RTCP.packet(ssrc=8, flost=9, clost=10, hseq=11, jitter=12, lsr=13, dlsr=14)
>>> rr = RTCP.packet(pt=RTCP.RR, ssrc=1, reports=[r1, r2])
```

As you can see, the `RTCP.packet` class can be used for many purposes. It defines dynamic attribute as well as container syntax for the properties, similar to the SDP class implemented in earlier chapter.

The RTCP SDES packet can be created as follows. Each item is a tuple, with a list of attributes such as CNAME, NAME, PHONE, etc.

```
>>> item1 = (1, [(RTCP.CNAME, 'kundan@example.net'), (RTCP.NAME, 'Kundan Singh'), (RTCP.EMAIL,
'kundan@example.net'), (RTCP.PHONE, '9176216392')])
>>> item2 = (2, [(RTCP.CNAME, 'henning@example.net'), ])
>>> sdes = RTCP.packet(pt=RTCP.SDES, items=[item1, item2])
```

An RTCP BYE packet can be created as follows.

```
>>> bye  = RTCP.packet(pt=RTCP.BYE, ssrcs=[1,2,3], reason='disconnecting')
```

The compound RTCP packet, with list semantics, can be created from these individual packets by supplying the list of individual packets.

```
>>> p1 = RTCP([sr, rr, sdes, bye])
```

For parsing an RTCP packet, you can construct the object using a single binary string argument. For example, we create `p2` by formatting and parsing back the original compound packet `p1`.

```
>>> p2 = RTCP(str(p1))
```

Let's walk through some more functions in RTCP packet. If you know the number of individual packets in the compound packet, you can use the list semantics on the compound packet to extract the individual packet. For example,

```
>>> sr, rr, sdes, bye = tuple(p2)
```

We can also access the various properties of the objects or sub-objects, either with attribute or with container access. Some examples follow, the results of which you can compare with the original values we set in our exercise.

```
>>> print sr.pt, sr.ssrc, sr.ntp, sr.ts, sr.pktcount, sr.octcount
200 1 2.0 3 4 5
>>> print rr.pt, rr.ssrc, [(x.ssrc, x.flost, x.clost, x.hseq, x.jitter, x.lsr, x.dlsr) for x in rr.reports]
201 1 [(1, 2, 3, 4, 5, 6, 7), (8, 9, 10, 11, 12, 13, 14)]
>>> print sdes.pt
202
>>> for item in sdes.items:
...    print 'ssrc=', item[0]
...    for n,v in item[1]: print ",n,'=',v
```

```
ssrc= 1
 1 = kundan@example.net
 2 = Kundan Singh
 3 = kundan@example.net
 4 = 9176216392
ssrc= 2
 1 = henning@example.net
>>> print bye.pt, bye.ssrcs, bye.reason
203 [1, 2, 3] disconnecting
```

Properties

Let's implement the `RTCP` class as a sub-class of `list`, so that it inherits the list semantics and syntax for representing the compound packet.

```
class RTCP(list):
```

The packet types and attribute types are defines are constants as per the specification.

```
    SR, RR, SDES, BYE, APP = range(200, 205) # various packet types
    CNAME, NAME, EMAIL, PHONE, LOC, TOOL, NOTE, PRIV = range(1, 9)
```

The nested class `packet` is similar to our `attrs` class in module `rfc4566`. It is used as a generic class for individual packet or report. It exposes both container and attribute interface. The construction can be done by supplying the named parameters. The attribute names are case-sensitive, unlike `attrs`.

```
    class packet(object):
        def __init__(self, **kwargs):
            for n,v in kwargs.items(): self[n] = v
        def __getattr__(self, name): return self.__getitem__(name)
        def __getitem__(self, name): return self.__dict__.get(name, None)
        def __setitem__(self, name, value): self.__dict__[name] = value
        def __contains__(self, name): return name in self.__dict__
```

Parsing

If a `value` is supplied to the constructor of RTCP object, and the `value` is a list, then it just gets appended to the compound packet. It assumes that the list contained the individual RTCP packets.

```
    def __init__(self, value=None): # parse the compound RTCP packet.
        if isinstance(value, list):
```

```
        for v in value: self.append(v) # just append the list of packets
        return
```

Otherwise, we parse each individual packet from the binary string `value`.

```
    while value and len(value)>0:
        p = RTCP.packet() # individual RTCP packet
```

The first four-bytes contain the necessary information such as version number, packet type and length. We validate for the version and packet type and throw an error if they are invalid.

```
        px, p.pt, plen = struct.unpack('!BBH', value[:4])
        if px & 0xC0 != 0x80: raise ValueError, 'RTP version must be 2'
        if p.pt < 200 or p.pt >= 205: raise ValueError, 'Not an RTCP packet type %d'%(p.pt)
```

Based on the length extracted earlier, we extract the current packet, remove the optional padding, and then based on the packet type perform further parsing.

```
        data, value = value[4:4+plen*4], value[4+plen*4:] # data for this packet, value for next
        if px & 0x20: data = data[:len(data)-ord(data[-1])] # remove padding
```

The sender report and receiver report have initial headers followed by list of report items. Each report item is parsed into `RTCP.packet` item.

```
        if p.pt == RTCP.SR or p.pt == RTCP.RR:
            if p.pt == RTCP.SR:
                p.ssrc, ntp1, ntp2, p.ts, p.pktcount, p.octcount = struct.unpack('!IIIIII', data[:24])
                p.ntp = ntp2time((ntp1, ntp2))
                data = data[24:]
            else:
                p.ssrc, = struct.unpack('!I', data[:4])
                data = data[4:]
            p.reports = []
            for i in range(px&0x1f):
                r = RTCP.packet()
                r.ssrc, lost, r.hseq, r.jitter, r.lsr, r.dlsr = struct.unpack('!IIIIII', data[:24])
                r.flost, r.clost = (lost >> 24) & 0x0ff, (lost & 0x0ffffff)
                p.reports.append(r)
                data = data[24:]
            p.extn = data if data else None
```

The source description packet has list of items, where each item describes a single source (SSRC). Each item has list of description elements which are type-value tuples.

```
        elif p.pt == RTCP.SDES:
            p.items = []
            for i in range(0, px&0x1f):
                ssrc, = struct.unpack('!I', data[:4])
```

```
            items = []
            data, count = data[4:], 0
            while len(data)>0:
                itype, ilen = struct.unpack('!BB', data[:2])
                count += (2 + ilen)
                ivalue, data = data[2:2+ilen], data[2+ilen:]
                if itype == 0: break
                items.append((itype, ivalue))
            if count % 4 != 0: data = data[(4-count%4):] # ignore padding for the chunk
            p.items.append((ssrc, items))
```

The termination (BYE) packet has list of sources that are terminating the session.

```
        elif p.pt == RTCP.BYE:
            p.ssrcs, p.reason = [], None
            for i in range(0, px & 0x01f):
                ssrc, = struct.unpack('!I', data[:4])
                p.ssrcs.append(ssrc)
                data = data[4:]
            if data and len(data)>0:
                rlen, = struct.unpack('!B', data[:1])
                p.reason = data[1:1+rlen] # no need to ignore padding, it already gets ignored when we use next packet
```

The application defined packet the source identifier, name and data.

```
        elif p.pt == RTCP.APP:
            p.subtype = px&0x1f
            p.ssrc, p.name = struct.unpack('!I4s', data[:8])
            p.data = data[8:]
            if not p.data: p.data = None
```

Any other packet type's raw data is stored as it is.

```
        else: # just store the raw data
            p.subtype = px&0x1f
            p.data = data[4:]
```

Once an individual packet is parsed, it is appended to the RTCP compound packet list.

```
        self.append(p)
```

Formatting

Formatting a compound RTCP packet iterates over all the packets and formats them. The pack and unpack operations for parsing and formatting are dual of each other.

```
    def __str__(self):
        result = ''
        for p in self:
            count, value = 0, ''
            if p.pt == RTCP.SR or p.pt == RTCP.RR:
                if p.pt == RTCP.SR:
                    ntp1, ntp2 = time2ntp(p.ntp)
                    value = struct.pack('!IIIIII', p.ssrc, ntp1, ntp2, p.ts, p.pktcount, p.octcount)
                else: value = struct.pack('!I', p.ssrc)
                count = len(p.reports)
                for r in p.reports:
                    value += struct.pack('!IIIIII', r.ssrc, (r.flost << 24) | (r.clost & 0x0ffffff), r.hseq, r.jitter, r.lsr, r.dlsr)
                if p.extn: value += p.extn
            elif p.pt == RTCP.SDES:
                count = len(p.items)
                for ssrc,items in p.items:
                    chunk = struct.pack('!I', ssrc)
                    for n,v in items:
                        chunk += struct.pack('!BB', n, len(v)>255 and 255 or len(v)) + v[:256]
                    chunk += struct.pack('!BB', 0, 0) # to indicate end of items.
                    if len(chunk)%4!=0: chunk += '\x00'*(4-len(chunk)%4)
                    value += chunk
            elif p.pt == RTCP.BYE:
                count = len(p.ssrcs)
                for ssrc in p.ssrcs: value += struct.pack('!I', ssrc)
                if p.reason and len(p.reason)>0: value += struct.pack('!B', len(p.reason)>255 and 255 or len(p.reason)) +
p.reason[:256]
            elif p.pt == RTCP.APP:
                count = p.subtype
                value += struct.pack('!I4s', p.ssrc, p.name) + (p.data if p.data else '')
            else: # just add the raw data
                count = p.subtype
                value += p.data
            length = len(value)/4 + (1 if len(value)%4 != 0 else 0)
            result += struct.pack('!BBH', 0x80 | (len(value)%4 != 0 and 0x20 or 0x00) | (count & 0x1f), p.pt, length) \
                + value + ('' if (len(value) % 4 == 0) else ('\x00'*(4-len(value)%4-1) + struct.pack('!B', 4-len(value)%4)))
        # TODO: we do padding in each packet, instead of only in last.
        return result
```

Constants

The specification defines the following constants that we need in our implementation.

```
RTP_SEQ_MOD   = (1<<16)
MAX_DROPOUT   = 3000
MAX_MISORDER  = 100
MIN_SEQUENTIAL = 2
```

Source

A source in a RTP-based session is implemented using the `Source` class. It represents both the local member as well as the remote members. The SSRC and SDES's CNAME must be unique in a session.

```
class Source(object):
```

Properties

We can create a new member for the given SSRC as follows.

```
>>> m = Source(1, [(RTCP.CNAME, 'kundan@example.net'), (RTCP.NAME, 'Kundan Singh')], ('127.0.0.1', 8000))
>>> print m
<Source ssrc=1 items=[(1, 'kundan@example.net'), (2, 'Kundan Singh')] address=('127.0.0.1', 8000) lost=0 fraction=0
pktcount=0 octcount=0 maxseq=0 badseq=0 cycles=0 baseseq=0 probation=0 received=0 expectedprior=0
receivedprior=0 transit=0 jitter=0 lastts=None lastntp=None rtcpdelay=None>
```

The specification defines various properties for a source, which we use in our implementation. We also have additional properties as needed in our implementation.

From RFC3550 p.78 –

```
/*
 * Per-source state information
 */
typedef struct {
  u_int16 max_seq;        /* highest seq. number seen */
  u_int32 cycles;         /* shifted count of seq. number cycles */
  u_int32 base_seq;       /* base seq number */
  u_int32 bad_seq;        /* last 'bad' seq number + 1 */
  u_int32 probation;      /* sequ. packets till source is valid */
  u_int32 received;       /* packets received */
  u_int32 expected_prior; /* packet expected at last interval */
  u_int32 received_prior; /* packet received at last interval */
  u_int32 transit;        /* relative trans time for prev pkt */
  u_int32 jitter;         /* estimated jitter */
  /* ... */
} source;
```

```
  def __init__(self, ssrc, items=[], address=None):
    self.ssrc, self.items, self.address = ssrc, items, address
    self.lost = self.fraction = self.pktcount = self.octcount = self.timeout = 0
    self.maxseq = self.badseq = self.cycles = self.baseseq = self.probation = self.received = self.expectedprior =
self.receivedprior = self.transit = self.jitter = 0 # based on RFC 3550's source structure
    self.lastts = self.lastntp = self.rtcpdelay = None
```

The string representation is used for debugging purpose, which prints all the attributes of the object.

```
def __repr__(self):
   props = ('ssrc', 'items', 'address', 'lost', 'fraction', 'pktcount', 'octcount', \
         'maxseq', 'badseq', 'cycles', 'baseseq', 'probation', 'received',     \
         'expectedprior', 'receivedprior', 'transit', 'jitter', 'lastts',    \
         'lastntp', 'rtcpdelay')
   return ('<Source ' + ' '.join([p+'=%r' for p in props]) + '>')%tuple([(eval('self.%s'%p)) for p in props])
```

Initializing sequence

When a new RTP packet is received, the sequence number of the received packet is used to initialize the sequence number of the source. The specification defines this as follows, which we port to our implementation.

```
void init_seq(source *s, u_int16 seq)
{
   s->base_seq = seq;
   s->max_seq = seq;
   s->bad_seq = RTP_SEQ_MOD + 1;   /* so seq == bad_seq is false */
   s->cycles = 0;
   s->received = 0;
   s->received_prior = 0;
   s->expected_prior = 0;
   /* other initialization */
}
```

```
def initseq(self, seq):
   self.baseseq = self.maxseq = seq
   self.badseq = seq - 1
   self.cycles = self.received = self.receivedprior = self.expectedprior = 0
   return self
```

This method can be tested as follows.

```
>>> print Source(ssrc=1).initseq(10)
<Source ssrc=1 items=[] address=None lost=0 fraction=0 pktcount=0 octcount=0 maxseq=10 badseq=9 cycles=0
baseseq=10 probation=0 received=0 expectedprior=0 receivedprior=0 transit=0 jitter=0 lastts=None lastntp=None
rtcpdelay=None>
```

Newly found source

When a new source is heard for the first time, that is, its SSRC identifier is not in the table, and the per-source state is allocated for it, s->probation is set to the number of sequential packets required before declaring a source valid (parameter MIN_SEQUENTIAL) and other variables are initialized:

```
init_seq(s, seq);
s->max_seq = seq - 1;
s->probation = MIN_SEQUENTIAL;
```

A non-zero s->probation marks the source as not yet valid so the state may be discarded after a short timeout rather than a long one.

```
def newfound(self, seq):
    self.initseq(seq)
    self.maxseq, self.probation = seq-1, MIN_SEQUENTIAL
    return self
```

This method can be tested as follows.

```
>>> print Source(ssrc=1).newfound(10)
<Source ssrc=1 items=[] address=None lost=0 fraction=0 pktcount=0 octcount=0 maxseq=9 badseq=9 cycles=0
baseseq=10 probation=2 received=0 expectedprior=0 receivedprior=0 transit=0 jitter=0 lastts=None lastntp=None
rtcpdelay=None>
```

Updating sequence on received packets

The specification defines the following C function to update the source properties based on the received RTP packet's sequence number. We port this to our implementation below.

```
int update_seq(source *s, u_int16 seq)
{
  u_int16 udelta = seq - s->max_seq;
  const int MAX_DROPOUT = 3000;
  const int MAX_MISORDER = 100;
  const int MIN_SEQUENTIAL = 2;

  /*
   * Source is not valid until MIN_SEQUENTIAL packets with
   * sequential sequence numbers have been received.
   */
  if (s->probation) {
    /* packet is in sequence */
    if (seq == s->max_seq + 1) {
      s->probation--;
      s->max_seq = seq;
      if (s->probation == 0) {
        init_seq(s, seq);
        s->received++;
        return 1;
      }
    } else {
      s->probation = MIN_SEQUENTIAL - 1;
      s->max_seq = seq;
    }
    return 0;
  } else if (udelta < MAX_DROPOUT) {
    /* in order, with permissible gap */
    if (seq < s->max_seq) {
```

```
        /*
         * Sequence number wrapped - count another 64K cycle.
         */
        s->cycles += RTP_SEQ_MOD;
      }
      s->max_seq = seq;
    } else if (udelta <= RTP_SEQ_MOD - MAX_MISORDER) {
      /* the sequence number made a very large jump */
      if (seq == s->bad_seq) {
        /*
         * Two sequential packets -- assume that the other side
         * restarted without telling us so just re-sync
         * (i.e., pretend this was the first packet).
         */
        init_seq(s, seq);
      }
      else {
        s->bad_seq = (seq + 1) & (RTP_SEQ_MOD-1);
        return 0;
      }
    } else {
      /* duplicate or reordered packet */
    }
    s->received++;
    return 1;
}
```

```
def updateseq(self, seq):
    udelta = seq - self.maxseq
    if self.probation > 0:
        if seq == self.maxseq+1:
            self.probation, self.maxseq = self.probation - 1, seq
            if self.probation == 0:
                self.initseq(seq)
                self.received = self.received + 1
                return self # True
        else:
            self.probation, self.maxseq = MIN_SEQUENTIAL-1, seq # at least next one packet should be in sequence
        return self # False
    elif udelta < MAX_DROPOUT: # in order, with permissible gap
        if seq < self.maxseq: self.cycles += RTP_SEQ_MOD
        self.maxseq = seq
    elif udelta <= RTP_SEQ_MOD - MAX_MISORDER: # the seq made a very large jump
        if seq == self.badseq: self.initseq(seq) # probably the other side reset the seq
        else:
            self.badseq = (seq + 1) & (RTP_SEQ_MOD-1)
            return self # False
    self.received = self.received + 1
    return self # True
```

This can be tested as follows.

```
>>> print Source(1).newfound(10).updateseq(12).updateseq(13) # simulate loss of 11
```

```
<Source ssrc=1 items=[] address=None lost=0 fraction=0 pktcount=0 octcount=0 maxseq=13 badseq=12 cycles=0
baseseq=13 probation=0 received=1 expectedprior=0 receivedprior=0 transit=0 jitter=0 lastts=None lastntp=None
rtcpdelay=None>
```

Estimating the interarrival jitter

The code fragments below implement the algorithm given in Section 6.4.1 for calculating an estimate of the statistical variance of the RTP data interarrival time to be inserted in the interarrival jitter field of reception reports. The inputs are r->ts, the timestamp from the incoming packet, and arrival, the current time in the same units. Here s points to state for the source; s->transit holds the relative transit time for the previous packet, and s->jitter holds the estimated jitter. The jitter field of the reception report is measured in timestamp units and expressed as an unsigned integer, but the jitter estimate is kept in a floating point. As each data packet arrives, the jitter estimate is updated:

```
int transit = arrival - r->ts;
int d = transit - s->transit;
s->transit = transit;
if (d < 0) d = -d;
s->jitter += (1./16.) * ((double)d - s->jitter);
```

When a reception report block (to which rr points) is generated for this member, the current jitter estimate is returned:

```
rr->jitter = (u_int32) s->jitter;
```

The following method updates the jitter based on the timestamp `ts` and `arrival` timestamp (also in `ts` units).

```
def updatejitter(self, ts, arrival):
    transit = int(arrival - ts)
    d, self.transit = int(math.fabs(transit - self.transit)), transit
    self.jitter += (1/16.) * (d-self.jitter)
    return self
```

This can be tested as follows

```
>>> s = Source(1).newfound(10).updatejitter(1000, 0).updatejitter(1160, 160).updatejitter(1330, 320)
>>> print s.transit, int(s.jitter)
-1010 55
```

Determining the number of packets expected and lost

In order to compute packet loss rates, the number of RTP packets expected and actually received from each source needs to be known, using per-source state information defined in struct source referenced via pointer s in the code below. The number of packets received is simply the count of packets as they arrive, including any late or duplicate packets. The number of packets expected can be computed by the receiver as the difference between the highest sequence number received (s->max_seq) and the first sequence number received (s->base_seq). Since the sequence number is only 16 bits and will wrap around, it is necessary to extend the

highest sequence number with the (shifted) count of sequence number wraparounds (s->cycles). Both the received packet count and the count of cycles are maintained the RTP header validity check routine in Appendix A.1.

```
extended_max = s->cycles + s->max_seq;
expected = extended_max - s->base_seq + 1;
```

The number of packets lost is defined to be the number of packets expected less the number of packets actually received:

```
lost = expected - s->received;
```

Since this signed number is carried in 24 bits, it should be clamped at 0x7fffff for positive loss or 0x800000 for negative loss rather than wrapping around.

The fraction of packets lost during the last reporting interval (since the previous SR or RR packet was sent) is calculated from differences in the expected and received packet counts across the interval, where expected_prior and received_prior are the values saved when the previous reception report was generated:

```
expected_interval = expected - s->expected_prior;
s->expected_prior = expected;
received_interval = s->received - s->received_prior;
s->received_prior = s->received;
lost_interval = expected_interval - received_interval;
if (expected_interval == 0 || lost_interval <= 0) fraction = 0;
else fraction = (lost_interval << 8) / expected_interval;
```

The resulting fraction is an 8-bit fixed point number with the binary point at the left edge.

```
def updatelostandexpected(self):
    extendedmax = self.cycles + self.maxseq
    expected = extendedmax - self.baseseq + 1
    self.lost = expected - self.received
    expectedinterval = expected - self.expectedprior
    self.expectedprior = expected
    receivedinterval = self.received - self.receivedprior
    self.receivedprior = self.received
    lostinterval = expectedinterval - receivedinterval
    if expectedinterval == 0 or lostinterval <= 0: self.fraction = 0
    else: self.fraction = (lostinterval << 8) / expectedinterval
    return self
```

This can be tested as follows.

```
>>> s = Source(1).newfound(10).updateseq(11).updateseq(12).updateseq(14).updatelostandexpected() # loss of 13
>>> print s.lost, s.fraction, s.expectedprior, s.receivedprior
1 85 3 2
```

To store the report properties in the `Source` object, the `storereport` method can be invoked.

```
def storereport(self, fraction, lost, jitter, delay):
    self.fraction, self.lost, self.jitter, self.rtcpdelay = fraction, lost, jitter, delay
    return self
```

Timestamp conversion

The specification heavily uses the Network Time Protocol (NTP) time format. The NTP time has higher resolution, and a different offset than the UNIX time that many operating systems provide. The offset can be found in the NTP specification.

We define two methods: `time2ntp` and `ntp2time`, to convert between the UNIX time format and NTP time format. Python's `time.time()` method gives the current time UNIX time format. We use a tuple of two values to represent NTP time format. The first value is the number of seconds and the second value is the fraction. These values are readily usable in our RTP and RTCP implementation wherever NTP time format is desired.

To convert the UNIX time 0.5 to NTP you can do the following.

```
>>> print time2ntp(0.5)
(2208988800L, 2147483648L)
```

Similarly to convert from NTP time tuple to the UNIX time format you can do the following.

```
>>> print ntp2time(time2ntp(10.5))
10.5
```

Once we know the offset and conversion factor, the conversion is straightforward as defined below.

```
def time2ntp(value):
    value = value + 2208988800
    return (int(value), int((value-int(value)) * 4294967296.))
```

```
def ntp2time(value):
    return (value[0] + value[1] / 4294967296.) - 2208988800
```

RTP Session

From RFC3550 p.9 – RTP session: An association among a set of participants communicating with RTP. A participant may be involved in multiple RTP sessions at the same time. In a multimedia session, each medium is typically carried in a separate RTP session with its own RTCP packets unless the the encoding itself multiplexes multiple media into a single data stream. A participant distinguishes multiple RTP sessions by reception of different sessions using different pairs of destination transport addresses, where a pair of transport addresses comprises one network address plus a pair of ports for RTP and RTCP. All participants in an RTP session may

share a common destination transport address pair, as in the case of IP multicast, or the pairs may be different for each participant, as in the case of individual unicast network addresses and port pairs. In the unicast case, a participant may receive from all other participants in the session using the same pair of ports, or may use a distinct pair of ports for each.

The distinguishing feature of an RTP session is that each maintains a full, separate space of SSRC identifiers (defined next). The set of participants included in one RTP session consists of those that can receive an SSRC identifier transmitted by any one of the participants either in RTP as the SSRC or a CSRC (also defined below) or in RTCP. For example, consider a three-party conference implemented using unicast UDP with each participant receiving from the other two on separate port pairs. If each participant sends RTCP feedback about data received from one other participant only back to that participant, then the conference is composed of three separate point-to-point RTP sessions. If each participant provides RTCP feedback about its reception of one other participant to both of the other participants, then the conference is composed of one multi-party RTP session. The latter case simulates the behavior that would occur with IP multicast communication among the three participants.

The RTP framework allows the variations defined here, but a particular control protocol or application design will usually impose constraints on these variations.

Application interface

Let's assume that the Session class implements an RTP session and presents the high level application interface. The application installs itself in the session when constructing a new session. The session object invokes various callbacks on the application to notify important events. The application first constructs the session object then starts the session. When the session is completed it must stop the session.

```
session = Session(self, ...)  # install by supplying itself to session
session.start()
...
session.stop()
```

When the session is starting or stopping, it invokes the appropriate callbacks on the application, hence the application can implement the following to get notified of these events.

```
def starting(self, session): ...
def stopping(self, session): ...
```

Network interface: The application also implements the network interface. The session invokes the following callback when it actually wants to send some data on the RTP or RTCP socket. The application should use the network interface to actually send these.

```
def sendRTP(self, data): ...
def sendRTCP(self, data): ...
```

Similarly, when the application receives some data from the network interface, it invokes the appropriate methods on the session to transfer the data to the session.

```
session.receivedRTP(data, src, dest)
session.receivedRTCP(data, src, dest)
```

Media data: When the application wants to send some media data in the session, it invokes the send methods by supplying the media payload, and optional timestamp, marker and payload type arguments. Since the media source is the best place to keep the timestamp information, we let the application supply the timestamp to the session.

```
session.send(payload, timestamp, marker, payloadType)
```

When the session has some RTP packet, it invokes the following callback to inform the application about the packet. The application can access the various RTP headers including the timestamp and payload information.

```
def received(self, member, packet):
  print "payload length", len(packet.payload)
  ... # process the packet
```

The `member` argument is of type Source and identifies the source of the packet. The packet argument is of type `RTP`.

Timer: The session needs a timer implementation. However, to keep the software architecture independent of any asynchronous activity, we delegate the timer implementation to the application. This is similar to the motivation we used in the SIP library implementation earlier. Hence the application must implement the `createTimer` method to supply a timer object when the session demands it.

```
def createTimer(self, app):
  ... # see the timer in the "Session Initiation Protocol" chapter.
```

Now that we have defined the example usage of the session interface, which forms the high level interface of the RTP implementation, let's move on to the implementation of the `Session` class.

Properties

The `Session` class which implements the RTP session is the main control implementation to manager a single session. We define the following properties in a session. The optional payload type, `pt`, and sampling rate, `rate`, control the payload type and sampling rate of the outgoing RTP packet. The default value of `pt` is 96 representing a dynamic payload type, and that of `rate` is 8000 Hz. The `bandwidth` property specifies the total session bandwidth and defaults to 64000 (bits/second). The `fraction` property specifies the fraction of bandwidth to use for RTCP and defaults to 0.05 indicating 5% of the total bandwidth. The optional `member` property refers to the `Source` object for this member. By default it constructs a new source member. The `ssrc` and `cname` properties are useful only when constructing a new source member, instead of using a random number and name. The `seq0` and `ts0` optional properties control the initial sequence number and timestamp in outgoing RTP packets, instead of using random numbers by default.

The constructor can take these properties as optional named arguments. The first argument is a reference to the application, so that the RTP implementation can invoke callbacks to signal events to the application.

```
class Session(object):
  def __init__(self, app, **kwargs):
    self.app, self.pt, self.rate, self.bandwidth, self.fraction, self.member    = \
      app, kwargs.get('pt', 96), kwargs.get('rate', 8000), kwargs.get('bandwidth', 64000), kwargs.get('fraction', 0.05),
kwargs.get('member', None)
    if not self.member:
      ssrc  = kwargs.get('ssrc', random.randint(0, 2**32))
      cname = kwargs.get('cname', '%d@%s'%(ssrc, getlocaladdr()))
      self.member = Source(ssrc=ssrc, items=[(RTCP.CNAME, cname)])
    self.seq0, self.ts0 = kwargs.get('seq0', self.randint(0, 2**16)), kwargs.get('ts0', self.randint(0, 2**32))
    self.seq = self.ts = self.ts1 = 0 # recent seq and ts. ts1 is base time.
    self.ntp = self.ntp1 = self.tc    # recent NTP time and base time.
```

When constructing a session we also initialize some statistics as below.

```
    self.rtpsent = self.rtcpsent = self.byesent = self.running = False
```

From RFC3550 p.29 – A session participant must maintain several pieces of state:

tp: the last time an RTCP packet was transmitted;
tc: the current time;
tn: the next scheduled transmission time of an RTCP packet;
pmembers: the estimated number of session members at the time tn was last recomputed;
members: the most current estimate for the number of session members;
senders: the most current estimate for the number of senders in the session;
rtcp_bw: The target RTCP bandwidth, i.e., the total bandwidth that will be used for RTCP packets by all members of this session, in octets per second. This will be a specified fraction of the "session bandwidth" parameter supplied to the application at startup.
we_sent: Flag that is true if the application has sent data since the 2nd previous RTCP report was transmitted.
avg_rtcp_size: The average compound RTCP packet size, in octets, over all RTCP packets sent and received by this participant. The size includes lower-layer transport and network protocol headers (e.g., UDP and IP).
initial: Flag that is true if the application has not yet sent an RTCP packet.

```
    self.tp = self.tn = 0 # tp=last RTCP transmit time, tc=current time, tn=next RTCP scheduled time
    self.members, self.senders = dict(), dict()  # TODO: this should be a smart set+map data structure
    self.pmembers = 0
    self.rtcpbw = self.bandwidth*self.fraction
    self.wesent, self.initial, self.avgrtcpsize = False, True, 200
```

The `randint` method generates a random number in the given range, which defaults to the four-bytes number range. TODO: we should modify this to use the algorithm as defined in the RFC instead of using the `random` module.

```
  def randint(self, low=0, high=0x100000000):
    return random.randint(low, high) # Return a random number between [low, high).
```

The `tc` read-only property returns the current UNIX time in double.

```
    @property
    def tc(self):
        return time.time()
```

The `tsnow` read-only property returns the current time in RTP timestamp unit.

```
    @property
    def tsnow(self):
        return int(self.ts + (self.tc - self.ntp)*((self.ts - self.ts1) / (self.ntp - self.ntp1))) & 0xffffffff
```

Starting and stopping

We implement two methods, `start` and `stop`. The application can invoke these methods to control the session. The session is set to be in `running` state when started, until it is stopped. A running session receives incoming packets, and periodically sends outgoing RTCP packets. The sending of RTP packets is controlled by the application.

Calling the `start` method on an already running session has no effect.

```
    def start(self):
        if self.running: return # already running, don't run again.
```

When a session is started, we clear its membership state and packets statistics.

```
        self.senders.clear(); self.members.clear(); # add ourself in members.
        self.pmembers = 1
        self.members[self.member.ssrc] = self.member
        self.wesent = self.rtcpsent = False
```

Then we schedule the timer for sending out the RTCP packets. The timeout is adjusted every time an RTCP packet is sent. The timeout handler schedules the next timer.

```
        delay = self.rtcpinterval(True) # compute first RTCP interval
        self.tp, self.tn = self.tc, self.tc + delay
        self.timer = timer = self.app.createTimer(self) # schedule a timer to send RTCP
        timer.start(delay*1000)
```

Finally, we set the state to be `running`, and inform the application that the session is starting by invoking `app.starting` method.

```
        self.running = True
```

```
        self.app.starting(self)
```

When the application wants to close the session, it invokes the `stop` method. This stops sending and receiving of the packets in this session. If the session is not already running then it has no effect.

```
    def stop(self, reason=''):
        if not self.running: return # not running already. Don't bother.
```

First we sent the RTCP BYE packet with the supplied `reason` property.

```
        sendBye(reason=reason)
```

Then we clear the membership state for this session.

```
        self.members.clear()
        self.senders.clear()
        self.pmembers = 0
```

Then we close any active timer for this session.

```
        if self.timer:
            self.timer.stop()
            self.timer = None
```

Finally we set the `running` state to be false, and inform the application by calling `app.stopping` method.

```
        self.running = False
        self.app.stopping(self)
```

Sending and receiving RTP

When the application wants to send some RTP packet in this session, it invokes the `send` method. The timestamp, marker and payload type can be set explicitly for each packet. The method just builds a new RTP object with the supplied parameters and invokes the application callback `app.sendRTP` to actually send the packet. It also updates the statistics as shown below.

```
    def send(self, payload='', ts=0, marker=False, pt=None):
        member = self.member
        member.pktcount = member.pktcount + 1
        member.octcount = member.octcount + len(payload)
        self.ts, self.ntp = ts, self.tc
```

```
    if self.ts1 == 0: self.ts1 = ts
    self.rtpsent = self.wesent = True

    if pt is None: pt = self.pt
    pkt = RTP(pt=pt, marker=marker, seq=self.seq0+self.seq, ts=self.ts0+ts, ssrc=member.ssrc, payload=payload)
    self.app.sendRTP(pkt)

    self.seq = self.seq + 1
```

When the network layer receives a new packet on the RTP port, the application should transfer the packet to the `Session` object by invoking the `receivedRTP` method. The raw received data along with the source and destination host-port tuples are supplied in this method. The method first parses the received data into an `RTP` object.

```
def receivedRTP(self, data, src, dest):
    p = RTP(data)
```

From RFC3550 p.31 – Receiving an RTP or Non-BYE RTCP Packet

When an RTP or RTCP packet is received from a participant whose SSRC is not in the member table, the SSRC is added to the table, and the value for members is updated once the participant has been validated as described in Section 6.2.1. The same processing occurs for each CSRC in a validated RTP packet.

When an RTP packet is received from a participant whose SSRC is not in the sender table, the SSRC is added to the table, and the value for senders is updated.

For each compound RTCP packet received, the value of avg_rtcp_size is updated:

avg_rtcp_size = (1/16) * packet_size + (15/16) * avg_rtcp_size

where packet_size is the size of the RTCP packet just received.

```
    member = None
    if p.ssrc not in self.members and self.running:
        member = self.members[p.ssrc] = Source(ssrc=p.ssrc).newfound(p.seq)
    elif self.running:
        member = self.members[p.ssrc]
    if p.ssrc not in self.senders and self.running:
        self.senders[p.ssrc] = self.members[p.ssrc]
    if member:
        member.received = member.received + 1
        member.timeout = 0
        member.address = src
        member.updateseq(p.seq)
        member.updatejitter(p.ts, self.tsnow)
```

Finally the method delivers the packet to the application by invoking the `app.received` callback.

```
        self.app.received(member, p)
```

Receiving RTCP

When the network layer receives a new packet on the RTCP port, the application should transfer the packet to the `Session` object by invoking the `receivedRTCP` method. The raw received data along with the source and destination host-port tuples are supplied in this method. The method first parses the received data into an `RTCP` compound object. Then it processes each individual packet.

```
def receivedRTCP(self, data, src, dest):
  for p in RTCP(data):  # for each individual packet
```

From RFC3550 p.92 –

```
void OnReceive(packet p,
               event e,
               int *members,
               int *pmembers,
               int *senders,
               double *avg_rtcp_size,
               double *tp,
               double tc,
               double tn)
{
  /* What we do depends on whether we have left the group, and are
   * waiting to send a BYE (TypeOfEvent(e) == EVENT_BYE) or an RTCP
   * report.  p represents the packet that was just received.  */

  if (PacketType(p) == PACKET_RTCP_REPORT) {
    if (NewMember(p) && (TypeOfEvent(e) == EVENT_REPORT)) {
       AddMember(p);
       *members += 1;
    }
    *avg_rtcp_size = (1./16.)*ReceivedPacketSize(p) +
         (15./16.)*(*avg_rtcp_size);
  } else if (PacketType(p) == PACKET_RTP) {
    if (NewMember(p) && (TypeOfEvent(e) == EVENT_REPORT)) {
      AddMember(p);
      *members += 1;
    }
    if (NewSender(p) && (TypeOfEvent(e) == EVENT_REPORT)) {
      AddSender(p);
      *senders += 1;
    }
  } else if (PacketType(p) == PACKET_BYE) {
    *avg_rtcp_size = (1./16.)*ReceivedPacketSize(p) +
       (15./16.)*(*avg_rtcp_size);

    if (TypeOfEvent(e) == EVENT_REPORT) {
      if (NewSender(p) == FALSE) {
         RemoveSender(p);
```

```
            *senders -= 1;
         }
         if (NewMember(p) == FALSE) {
            RemoveMember(p);
            *members -= 1;
         }

         if (*members < *pmembers) {
            tn = tc +
               (((double) *members)/(*pmembers))*(tn - tc);
            *tp = tc -
               (((double) *members)/(*pmembers))*(tc - *tp);

            /* Reschedule the next report for time tn */

            Reschedule(tn, e);
            *pmembers = *members;
         }

      } else if (TypeOfEvent(e) == EVENT_BYE) {
          *members += 1;
      }
   }
}
```

```
    if p.pt == RTCP.SR or p.pt == RTCP.RR:
       if p.ssrc not in self.members and self.running:
          self.members[p.ssrc] = Source(ssrc=p.ssrc)
       member = self.members[p.ssrc] # identify the member
       if p.pt == RTCP.SR:
          member.lastts  = p.ts
          member.lastntp = p.ntp
       member.timeout = 0
       for r in p.reports:
          if r.ssrc == self.member.ssrc:
             self.member.storereport(r.flost, r.clost, r.jitter, r.dlsr/65536.)
             break
    elif p.pt == RTCP.SDES:
       for ssrc,items in p.items:
          if ssrc not in self.members:
             member = self.members[ssrc] = Source(ssrc=ssrc)
          else:
             member = self.members[ssrc]
          member.items = items # override previous items list
```

From RFC3550 p.31 – Except as described in Section 6.3.7 for the case when an RTCP BYE is to be transmitted, if the received packet is an RTCP BYE packet, the SSRC is checked against the member table. If present, the entry is removed from the table, and the value for members is updated. The SSRC is then checked against the sender table. If present, the entry is removed from the table, and the value for senders is updated.

Furthermore, to make the transmission rate of RTCP packets more adaptive to changes in group membership, the following "reverse reconsideration" algorithm SHOULD be executed when a BYE packet is received that reduces members to a value less than pmembers:

- The value for tn is updated according to the following formula:

tn = tc + (members/pmembers) * (tn - tc)

- The value for tp is updated according the following formula:

  tp = tc - (members/pmembers) * (tc - tp).

- The next RTCP packet is rescheduled for transmission at time tn, which is now earlier.

- The value of pmembers is set equal to members.

This algorithm does not prevent the group size estimate from incorrectly dropping to zero for a short time due to premature timeouts when most participants of a large session leave at once but some remain. The algorithm does make the estimate return to the correct value more rapidly. This situation is unusual enough and the consequences are sufficiently harmless that this problem is deemed only a secondary concern.

```
elif p.pt == RTCP.BYE:
    for ssrc in p.ssrcs:
        if ssrc in self.members:
            del self.members[ssrc]
        if ssrc in self.senders:
            del self.senders[ssrc]
        if self.running:
            self.timer.stop()
            self.tn = self.tc + (len(self.members)/self.pmembers) * (self.tn-self.tc)
            self.tp = self.tc - (len(self.members)/self.pmembers) * (self.tc-self.tp)
            self.timer.start((self.tn - self.tc) * 1000)
            self.pmembers = len(self.pmembers)
```

For each compound RTCP packet received, the value of avg_rtcp_size is updated:

avg_rtcp_size = (1/16) * packet_size + (15/16) * avg_rtcp_size

where packet_size is the size of the RTCP packet just received.

```
self.avgrtcpsize = (1/16.)*len(data) + (15/16.)*self.avgrtcpsize
```

Computing the RTCP transmission internal.

From RFC3550 p.29 – To maintain scalability, the average interval between packets from a session participant should scale with the group size. This interval is called the calculated interval. It is obtained by combining a number of the pieces of state described above. The calculated interval T is then determined as follows:

1. If the number of senders is less than or equal to 25% of the membership (members), the interval depends on whether the participant is a sender or not (based on the value of we_sent). If the participant is a sender (we_sent true), the constant C is set to the average RTCP packet size (avg_rtcp_size) divided by 25% of the RTCP bandwidth (rtcp_bw), and the constant n is set to the number of senders. If we_sent is not true, the constant C is set to the average RTCP packet size divided by 75% of the RTCP bandwidth. The constant n is set to the number of receivers (members - senders). If the number of senders is greater than 25%, senders and receivers are treated together. The constant C is set to the average RTCP packet size divided by the total RTCP bandwidth and n is set to the total number of members. As stated in Section 6.2, an RTP profile MAY specify that the RTCP bandwidth may be explicitly defined by two separate parameters (call them S and R) for those participants which are senders and those which are not. In that case, the 25% fraction becomes S/(S+R) and the 75% fraction becomes R/(S+R). Note that if R is zero, the percentage of senders is never greater than S/(S+R), and the implementation must avoid division by zero.

2. If the participant has not yet sent an RTCP packet (the variable initial is true), the constant Tmin is set to 2.5 seconds, else it is set to 5 seconds.
3. The deterministic calculated interval Td is set to max(Tmin, n*C).
4. The calculated interval T is set to a number uniformly distributed between 0.5 and 1.5 times the deterministic calculated interval.
5. The resulting value of T is divided by e-3/2=1.21828 to compensate for the fact that the timer reconsideration algorithm converges to a value of the RTCP bandwidth below the intended average.

This procedure results in an interval which is random, but which, on average, gives at least 25% of the RTCP bandwidth to senders and the rest to receivers. If the senders constitute more than one quarter of the membership, this procedure splits the bandwidth equally among all participants, on average.

```
def rtcpinterval(self, initial=False):
   if len(self.senders) < 0.25*len(self.members):
      if self.wesent: C, n = self.avgrtcpsize / (0.25*self.rtcpbw), len(self.senders)
      else: C, n = self.avgrtcpsize / (0.75*self.rtcpbw), len(self.members) - len(self.senders)
   else: C, n = self.avgrtcpsize / self.rtcpbw, len(self.members)
   return (min(initial and 2.5 or 5.0, n*C)) * (random.random() + 0.5) / 1.21828
```

Sendind RTCP

When the RTCP timeout expires, we send the compound RTCP packet as necessary.

From RFC3550 p.90 –

```
void OnExpire(event e,
              int    members,
              int    senders,
              double rtcp_bw,
              int    we_sent,
              double *avg_rtcp_size,
              int    *initial,
              time_tp   tc,
              time_tp   *tp,
              int    *pmembers)
{
  /* This function is responsible for deciding whether to send an
   * RTCP report or BYE packet now, or to reschedule transmission.
   * It is also responsible for updating the pmembers, initial, tp,
   * and avg_rtcp_size state variables.  This function should be
   * called upon expiration of the event timer used by Schedule().
   */

  double t;     /* Interval */
  double tn;    /* Next transmit time */

  /* In the case of a BYE, we use "timer reconsideration" to
   * reschedule the transmission of the BYE if necessary */
  if (TypeOfEvent(e) == EVENT_BYE) {
     t = rtcp_interval(members,
                       senders,
```

```
                        rtcp_bw,
                        we_sent,
                        *avg_rtcp_size,
                        *initial);
      tn = *tp + t;
      if (tn <= tc) {
        SendBYEPacket(e);
        exit(1);
      } else {
        Schedule(tn, e);
      }

    } else if (TypeOfEvent(e) == EVENT_REPORT) {
      t = rtcp_interval(members,
                        senders,
                        rtcp_bw,
                        we_sent,
                        *avg_rtcp_size,
                        *initial);
      tn = *tp + t;
      if (tn <= tc) {
        SendRTCPReport(e);
        *avg_rtcp_size = (1./16.)*SentPacketSize(e) +
            (15./16.)*(*avg_rtcp_size);
        *tp = tc;

        /* We must redraw the interval.  Don't reuse the
           one computed above, since its not actually
           distributed the same, as we are conditioned
           on it being small enough to cause a packet to
           be sent */

        t = rtcp_interval(members,
                          senders,
                          rtcp_bw,
                          we_sent,
                          *avg_rtcp_size,
                          *initial);

        Schedule(t+tc,e);
        *initial = 0;
      } else {
        Schedule(tn, e);
      }
      *pmembers = members;
    }
}
```

```
def timedout(self, timer):
    if not self.running: # need to send BYE
        delay = self.rtcpinterval()
        self.tn = self.tp + delay
        if self.tn <= self.tc:
            self.sendBYE()
        else:
            self.timer.start((self.tn - self.tc) * 1000)
```

```
    else: # need to send report
        delay = self.rtcpinterval()
        self.tn = self.tp + delay
        if self.tn <= self.tc:
            size = self.sendRTCP()
            self.avgrtcpsize = (1/16.)*size + (15/16.)*self.avgrtcpsize
            self.tp = self.tc
            delay = self.rtcpinterval()
            self.initial = False
        else:
            delay = self.tn - self.tc
        self.pmembers = len(self.members)
        self.timer.start(delay*1000) # restart the timer
```

The `sendBYE` method is invoked by periodic `timedout` callback to send the BYE packet along with other reports.

```
def sendBYE(self, reason=''):
    if self.rtpsent and self.rtcpsent:
        sendRTCP(True)
```

The `sendRTCP` method sends a compound RTCP packet containing various optional reports. The optional argument allows sending the BYE packet. The method returns the number of bytes of the compound packet sent. After constructing the various individual packets in the compound packet, it invokes a callback on the application to actually send the RTCP packet.

```
def sendRTCP(self, sendbye=False):
    reports = []
    toremove = []
    for member in self.members.values():
        if member.received > 0:
            ntp1, ntp2 = time2ntp(member.lastntp)
            lsr  = ((ntp1 & 0x0ffff) << 16) | ((ntp2 >> 16) & 0x0ffff)
            dlsr = int((self.tc - member.lastntp)*65536)
            member.updatelostandexpected()
            report = RTCP.packet(ssrc=member.ssrc, flost=member.fraction, clost=member.lost, \
                    hseq=member.cycles+member.maxseq, jitter=int(member.jitter), lsr=lsr, dlsr=dlsr)
            reports.append(report)
            member.received = 0
        if member.timeout == 5: # if no packet within five RTCP intervals
            toremove.append(member.ssrc) # schedule it to be removed
        else:
            member.timeout = member.timeout + 1
    if toremove: # remove all timedout members
        for ssrc in toremove: del self.members[ssrc]

    packet = RTCP()
    if self.wesent: # add a sender report
        p = RTCP.packet(pt=RTCP.SR, ntp=self.tc, ts=self.tsnow+self.ts0, pktcount=self.member.pktcount,  \
                        octcount=self.member.octcount, reports=reports[:32])
```

```
        self.wesent = False
    else:
        p = RTCP.packet(pt=RTCP.RR, reports=reports[:32])
    packet.append(p)

    if len(reports)>=32: # add additional RR if needed
        reports = reports[32:]
        while reports:
            p, reports = RTCP.packet(pt=RTCP.RR, reports=reports[:32]), reports[32:]
            packet.append(p)

    p = RTCP.packet(pt=RTCP.SDES, items=self.member.items)
    # add SDES. Should add items only every few packets, except for CNAME which is added in every.
    packet.append(p)

    if sendbye: # add a BYE packet as well
        p = RTCP.packet(pt=RTCP.BYE, ssrcs=[self.member.ssrc]) # Need to add a reason as well
        packet.append(p)

    data = str(packet) # format for network data
    self.app.sendRTCP(data) # invoke app to send the packet
    self.rtcpsent = True
    return len(data)
```

So far we have described the various functions as per the specification to handle RTP and RTCP. However, the actual transport of the packets was delegated to the application. In a real implementation the application will use another object to represent the network transport for the session. The application will then use this network transport to send and receive any RTP and RTCP packet for that session.

We implement an example `Network` class next.

Network

The network interface is tied to a single session. This is a simple network interface that allocates two consecutive UDP ports for RTP and RTCP. The useful properties of the `Network` class are `src` and `dest`, which are the host-port tuple representing the source and destination addresses. These addresses define the session, as they are the receiving transport addresses for the session. By default it uses some random consecutive port number within the specified range as mentioned below.

The default behavior is to use the RTCP port number as one more than the RTP port number. However, certain applications such as SIP-based telephony between users behind network address translators may change the port numbers to some arbitrary port numbers. Thus to facilitate this, we define two additional properties, `srcRTCP` and `destRTCP`, which explicitly allow setting the RTCP ports different from the default. Once the object is created, the `src` and `dest` properties cannot be changed.

The constructor can take several optional named arguments that control the behavior. The argument `src` is the host-port tuple for listening address. The host part should be set to “0.0.0.0” indicating any local interface. If the port is specified in this tuple then it is used for RTP listening port. If the argument is missing or the port is 0, then it allocates any even port in a pre-defined range of port numbers. The `srcRTCP` property can also be specified as a host-port tuple indicating a different port number than the default value of one more than the RTP port number. The `dest` and `destRTCP` optional arguments are used to specify the destination address

to send RTP and RTCP packets to. The argument supplied in the `sendRTP` or `sendRTCP` method overrides the property supplied in the constructor for individual sending.

The `maxsize` property controls the maximum packet size that can be sent on this network object. The app property refers to the application instance on which `receivedRTP` and `receivedRTCP` callbacks are invoked when some data is available on RTP and RTCP sockets, respectively.

If the `src` is not specified then it picks a random consecutive port number pair in the range 10000-65535, such that the RTP port is an even number and the RTCP port is the next odd number. The default range can be changed using the `low` and `high` named arguments. The condition that the RTP port should be an even number can also be relaxed by setting the `even` named argument to False. The number of retries for allocating the port pair defaults to 20. This can be set using the `retry` named argument.

```
class Network(object):
   def __init__(self, app, **kwargs):
      self.app    = app
      self.src    = kwargs.get('src', ('0.0.0.0', 0))
      self.dest   = kwargs.get('dest', None)
      self.srcRTCP= kwargs.get('srcRTCP', (self.src[0], self.src[1] and self.src[1]+1 or 0))
      self.destRTCP=kwargs.get('destRTCP', None)
      self.maxsize = kwargs.get('maxsize', 1500)
```

The most important step in constructing the `Network` object is to allocate the two UDP ports and associated bound sockets listening on those ports. If the `src` argument has a valid port number, then it is used for binding the two sockets. The second socket's port is one more than what is specified in `src`.

```
      if self.src[1] != 0:  # specified port
         try:
            s1 = socket.socket(type=socket.SOCK_DGRAM)
            s2 = socket.socket(type=socket.SOCK_DGRAM)
            s1.bind(self.src)
            s2.bind((self.srcRTCP)
         except:
            s1.close(); s2.close();
            s1 = s2 = None
```

If no valid port is supplied in the `src` argument, then it tries to allocate two random consecutive ports in the range 10000-65535. The range, the number of retries, and whether to care for even number for RTP port – these parameters can be controlled by the named arguments supplied in the constructor.

```
      else:
         retry = kwargs.get('retry', 20)   # number of retries to do
         low   = kwargs.get('low', 10000)  # the range low-high for picking port number
         high  = kwargs.get('high', 65535)
         even  = kwargs.get('even', True)  # means by default use even port for RTP
         while retry>0:
            s1 = socket.socket(type=socket.SOCK_DGRAM)
            s2 = socket.socket(type=socket.SOCK_DGRAM)
            # don't bind to any (0) to avoid collision in RTCP, where some OS will allocate same port for RTP for retries
```

```
            if even:
                port = random.randint(low, high) & 0x0fffe # should not use high+1?
            else:
                port = random.randint(low, high) | 0x00001
            try:
                s1.bind((self.src[0], port))
                s2.bind((self.src[1], port+1))
                self.src, self.srcRTCP = s1.getsockname(), s2.getsockname()
            except:
                s1.close(); s2.close();
                s1 = s2 = None
            retry = retry - 1
```

Once the sockets are created and bound, we store the sockets and schedule the listening tasks for those sockets.

```
        if s1 and s2:
            self.rtp, self.rtcp = s1, s2
            multitask.add(self.receiveRTP(s1))
            multitask.add(self.receiveRTCP(s2))
        else:
            raise ValueError, 'cannot allocate sockets'
```

Destroying the `Network` object involves closing the listening sockets.

```
    def __del__(self):
        if self.rtp: self.rtp.close(); self.rtp = None
        if self.rtcp: self.rtcp.close(); self.rtcp = None
        if self.app: self.app = None
```

The listening tasks receive packets in the listening sockets and transfer them to the application by invoking the `app.receivedRTP` or `app.receivedRTCP` callback methods as needed.

```
    def receiveRTP(self, sock):
        try:
            while True:
                data, remote = yield multitask.recvfrom(sock, self.maxsize)
                if self.app: self.app.receivedRTP(data, remote, self.src)
        except: pass
```

```
    def receiveRTCP(self, sock):
        try:
            while True:
                data, remote = yield multitask.recvfrom(sock, self.maxsize)
                if self.app: self.app.receivedRTCP(data, remote, self.src)
        except: pass
```

When the application wants to send some packet on RTP or RTCP socket, it invokes the `sendRTP` or `sendRTCP` method, respectively.

```
def sendRTP(self, data, dest=None):
    if self.rtp:
        yield multitask.sendto(self.rtp, data, dest or self.dest)
```

```
def sendRTCP(self, data, dest=None):
    if self.rtcp:
        yield multitask.sendto(self.rtcp, data, dest or self.dest)
```

The simple network interface implemented in this section is good enough for a simple client implementation that can communicate with remote over UDP. In later chapters we will extend this to support traversal through network address translators and firewalls.

RTP profile for audio and video

When RTP session is advertised in SDP message body passed in SIP INVITE or 200 OK response, the SDP specifies a particular profile to be used. The "RTP/AVP" profile as defined in RFC 3551 provides guidelines on the use of RTP and RTCP for various audio and video codecs. In particular, it defines static payload types for some existing and known audio and video codecs.

For the purpose of this implementation, we implement a module named `rfc3551`, which lists the pre-defined codecs as per RFC3551. This is a very simple module that contains two exported methods: `type` and `desc`, for converting a text description of a codec to the static payload type number and vice-versa, respectively.

For example, the specification defines payload type of 3 for narrowband GSM audio codec operating at 8000 Hz. Thus, you can test the two functions as follows.

```
>>> print type('GSM/8000')
3
>>> print desc(3)
('GSM', 8000, 1, 3, 'GSM/8000')
```

To get a list of all the static payload type definitions in this module, you can do the following.

```
>>> for x in range(0, len(_types)):
...     name, rate, count, pt, d = desc(x)
...     assert(pt == x)
```

```
...     if d: assert(x == type(d))
...     if d: print '%d=>%s'%(pt, d),
0=>PCMU/8000 3=>GSM/8000 4=>G723/8000 5=>DVI4/8000 6=>DVI4/16000 7=>LPC/8000 8=>PCMA/8000
9=>G722/8000 10=>L16/44100/2 11=>L16/44100 12=>QCELP/8000 13=>CN/8000 14=>MPA/90000 15=>G728/8000
16=>DVI4/11025 17=>DVI4/22050 18=>G729/8000 25=>CelB/90000 26=>JPEG/90000 28=>nv/90000
31=>H261/90000 32=>MPV/90000 33=>MP2T/90000 34=>H263/90000
```

In the implementation, we initialize a list `_types`, that contains a list of all the payload type descriptions up to a maximum value of static payload type.

```
# static types: arranged in rows 0-5, 6-10, 11-15, ...
_types = ["PCMU/8000/1", None, None, "GSM/8000/1", "G723/8000/1", "DVI4/8000/1", \
  "DVI4/16000/1", "LPC/8000/1", "PCMA/8000/1", "G722/8000/1", "L16/44100/2",    \
  "L16/44100/1", "QCELP/8000/1", "CN/8000/1", "MPA/90000/1", "G728/8000/1",     \
  "DVI4/11025/1", "DVI4/22050/1", "G729/8000/1", None, None,                    \
  None, None, None, None, "CelB/90000/1",                                       \
  "JPEG/90000/1",  None, "nv/90000/1", None, None,                              \
  "H261/90000/1", "MPV/90000/1", "MP2T/90000/1", "H263/90000/1"]
```

Then we define an internal method `_type2desc`, that takes the payload type number, extracts the description string, and extracts the four components of the description string in a tuple: (1) the name, (2) sampling rate, (3) channel count, and (4) description text.

```
def _type2desc(t):
    if _types[t]:
        name, srate, scount = _types[t].split('/')
        return (name, int(srate), int(scount), t, name + '/' + srate + ('' if scount == '1' else '/' + scount))
    else:
        return (None, None, None, t, None)
```

Then we construct another list, `_desc`, of all the descriptions tuple. We also create a list, `_lowers`, similar to `_types` but with all lower case entries, so that we can compare and index using this. Note that the description is case insensitive.

```
_desc   = map(_type2desc, range(0, len(_types)))
_lowers = [(x and x.lower() or None) for x in _types]
```

Then we define our interface method, `type`, which takes the description string and returns the payload type number or -1 if the payload type is not found in the table. The description string is of the form “name/rate” or “name/rate/count”.

```
type = lambda x:  _lowers.index(x.lower()) if x and (x.lower() in _lowers) \
        else ((_lowers.index(x.lower()+'/1') if x and ((x.lower()+'/1') in _lowers) else -1))
```

We also define the `desc` method that takes the payload type number and returns a tuple of (`name`, `rate`, `count`, `pt`, `text`) where `name` is the codec name string, `rate` is the sampling rate number, `count` is the

channel count number, `pt` is the payload type number and `text` is the description text string. If the payload type is not found in the table, it still returns the tuple where all except `pt` are `None`.

```
desc = lambda x: _desc[x] if x >=0 and x < len(_desc) else (None, None, None, x, None)
```

Touch-tone interface

Implementing DTMF transport as per RFC 2833 and RFC 2198

There are several ways to transfer key-pad input in an Internet audio call. At the high level the information can be sent over the signaling or the media path. Sending the key-pad input over the signaling path is usually discouraged. There are two ways to transport the information in the media path – either it can remain in the audio payload or can be detected and sent as special payload data. The problem with the first approach is that it loses quality because of the codecs. Hence we will implement only the second approach.

RFC2833 defines the payload format for the DTMF tone to be carried in an RTP session. RFC 2198 defines the redundant data format needed for using RFC2833-based transport of DTMF tones. We implement some of the functions from these specifications in our modules `rfc2833` and `rfc2198`, respectively.

RTP payload for DTMF digits

The payload format is a simple binary protocol packet that stores the digit number as shown below.

Let's define the `DTMF` class to represent the payload. The object exposes various properties such as `E`, `R`, `volume`, `duration` and `key` as defined in the specification. The `key` argument specifies the actual key-pad input. There are two ways to create a DTMF object – by providing these individual properties as named parameters or by supplying the `value` argument that contains the received payload to parse.

```
>>> key1 = DTMF(key=7)
>>> key2 = DTMF(value=str(key1))
```

The parsing is done by the constructor and the formatting using the string context. We need to use the `struct` module to parse and format the binary protocol.

```
import struct
```

```
class DTMF(object):
    def __init__(self, value=None, **kwargs):
        if not value:
            self.event = self.mapkey(kwargs.get('key', None))
            self.E     = kwargs.get('end', False)
            self.R     = False # reserved bit
            self.volume = kwargs.get('volume', 0)
            self.duration=kwargs.get('duration', 200)
        else:
            self.event, second, self.duration = struct.unpack('!BBH', value)
            self.E, self.R = (second & 0x80 != 0), False # ignore the reserved bit
```

```
        self.volume = second & 0x3f
```

```
    def __repr__(self):
        return struct.pack('!BBH', self.event, (self.E and 0x80 or 0x00) | (self.volume & 0x3f), self.duration)
```

The internal method to map the key number to the corresponding string representing the key is shown below.

```
    @staticmethod
    def mapkey(key):
        '''Convert a key to an event.'''
        if not key or len(key)!= 1: return 16 # either empty or not one char
        index = '0123456789*#ABCD'.find(key)
        if index>= 0: return index
        else: return 16
```

The `createDTMF` method is the main module function that takes a string containing sequence of DTMF digits, and returns a list of the payload `DTMF` objects representing those digits. The application is responsible for using the returned value and sending them in an RTP session by invoking `session.send` method of a `Session` object as described in the previous chapter.

```
def createDTMFs(keys):
    result = map(lambda x: DTMF(key=x), keys)
    if result: result[-1].E = True # last one has E set to True
    return result
```

RTP payload for redundant audio data

```
# Copyright (c) 2007, Kundan Singh. All rights reserved. See LICENSING for details.
# @implements RFC2198 (Redundant RTP payload)

'''
Implements RTP payload for redundant audio data as per RFC 2198.
'''

import struct

def createRedundant(packets):
    '''Create redundant payload using the individual RTP packets. The packets arg is assumed
    to be a list of tuples (pt, timestamp, payload). The first packet is assumed to be
    primary, and is put the last. All other packets are put in the same order'''
    hdr, data = '', ''
```

```
    for p in packets[1:]:
        hdr += struct.pack('!BHB', 0x80 | p[0], p[1] - packets[0][1], len(p[2]))
        data += p[2]
    if packets:
        hdr += struct.pack('!BHB', packets[0][0], packets[0][1], len(packets[0][2]))
        data += packets[0][2]
    return hdr + data

def parseRedundant(packet, ts):
    '''Parse a redundant payload and return the individual payloads. The first in the result
    is the primary payload. Each payload is tuple (pt, timestamp, payload). The ts of the
    original RTP packet should be supplied as well.'''
    all = []
    while packet:
        pt, = struct.unpack('!B', packet[:1])
        packet = packet[1:]
        if pt & 0x80:
            all.insert(0, (pt))
        else:
            tsoffset, len = struct.unpack('!HB', packet[:3])
            packet = packet[3:]
            all.append((pt & 0x7f, tsoffset, len))
    result = []
    for a in all[1:]: # for all secondary data
        data = (a[0], ts+a[1], packet[:a[2]])
        packet = packet[a[2]:]
        result.append(data)
    if all:
        result.insert(0, (all[0][0], ts, packet)) # put remaining data as primary
    return result
```