SIP SIMPLE client SDK

Developer Guide

http://sipsimpleclient.com

THIS DOCUMENT REFLECTS VERSION 0.19.0 RELEASED ON SEPTEMBER 16TH, 2011

ABSTRACT

SIP SIMPLE client SDK is a Software Development Kit for easy development of SIP end-points that support rich media like Audio, Instant Messaging, File Transfers, Desktop Sharing and Presence. Other media types can be easily added by using an extensible high-level API.

Revision 99, Date 11/11/11 9:52 AM

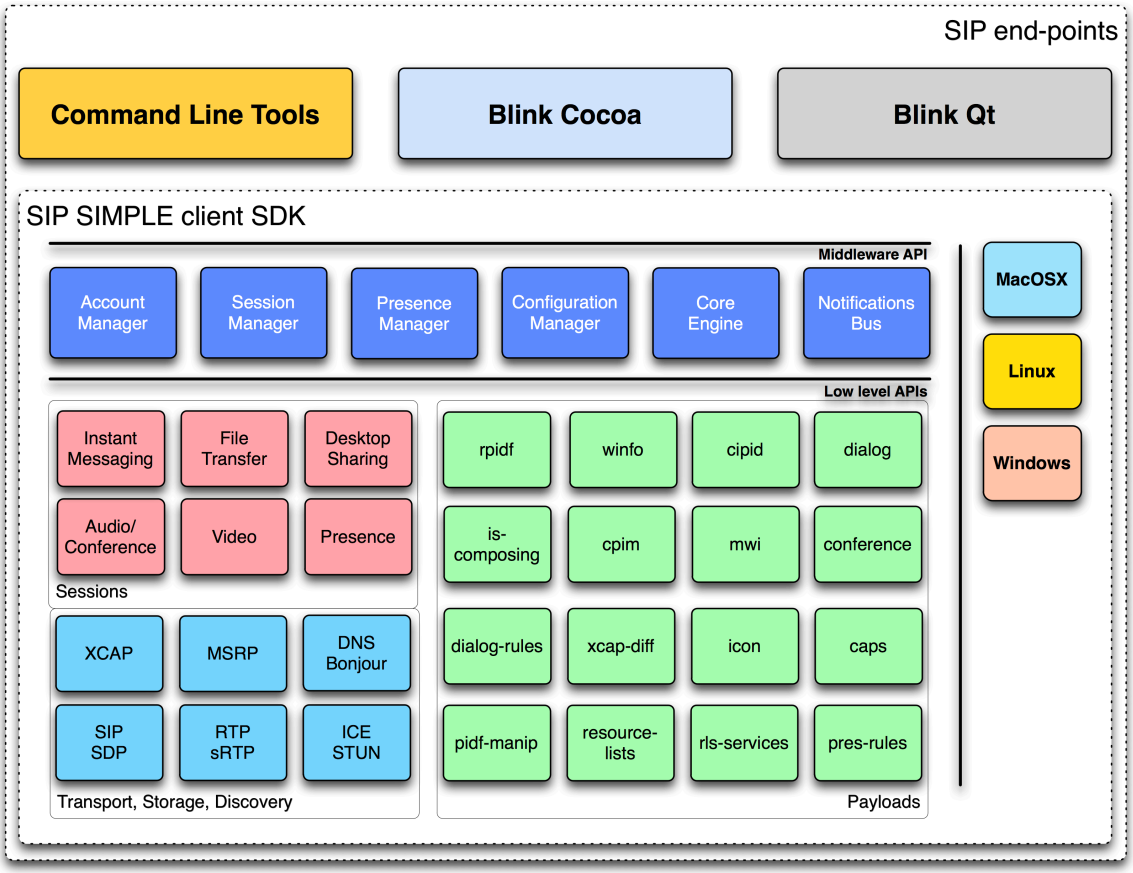
Description

SIP SIMPLE client SDK is a Software Development Kit for easy development of SIP end-points that support rich media like Audio, Instant Messaging, File Transfers, Desktop Sharing and Presence.

Other media types can be easily added by using an extensible high-level API.

The project home page is http://sipsimpleclient.com

Components



Background

SIP stands for 'Session Initiation Protocol', an IETF standard described by RFC 3261. SIP is an Internet application-layer control protocol that can establish, modify and terminate multimedia sessions such as Internet telephony calls (VoIP). Media can be added to (and removed from) an existing session.

SIP allows the endpoints to negotiate and combine any type of session they mutually understand like Audio, Video, Instant Messaging (IM), File Transfer, Desktop Sharing and provides a generic event notification system with real-time Publications and Subscriptions about state changes that can be used for asynchronous services like Presence, Message Waiting Indicator and Busy Line Appearance.

Features

General

- Multiple SIP Accounts support
- Non-blocking, asynchronous, notifications based engine
- Configuration Framework
- TLS Security for SIP signaling (SIP)
- TLS Security for media (MSRP, XCAP)
- Multiple Media Types per Session (e.g. Audio + IM)
- Trace capability for all underlying protocols
- Supports re-INVITE for adding and removing media
- Automatically handling if IP Address changes
- VPN friendly
- Conference Audio Mixer
- Wav Player and Recorder
- Acoustic Echo Cancelation
- Answering Machine
- Wide-band Internet codecs: Speex and G722
- PSTN compatible codecs: G711, iLBC, GSM

The SIP and media stacks are based on PJSIP 1.0 version with relevant patches from later versions applied.

Supported media types

- Audio (RTP/sRTP)
- Instant Messaging (MSRP)
- File Transfer (MSRP)
- Desktop Sharing (VNC over MSRP)

All media types can be combined together in the same SIP session.

Address Resolution

The library uses a separated from the core lookup mechanism for the next hop routing. This important feature allows the library to be used for building SIP clients that operate in combination with any SIP provider (by employing RFC 3263 DNS lookups), in a server-less LAN operation (using Bonjour protocol) or integrated into a network overlay developed by a third party (e.g. P2PSIP overlay).

Implemented Standards

SIP Signaling

- SIP, Session Initiation Protocol RFC3261
- SDP, Session Description Protocol RFC4566
- An Offer/Answer Model with Session Description Protocol (SDP) RFC3264
- Reliability of Provisional Responses in Session Initiation Protocol RFC3262
- HTTP Authentication: Basic and Digest Access Authentication RFC2617
- The Reason Header Field for the Session Initiation Protocol RFC3326

Address Resolution

1. Locating SIP services: RFC3263
2. Bonjour multicast DNS: http://tools.ietf.org/html/draft-lee-sip-dns-sd-uri-03

NAT Traversal

- SIP Signaling: Symmetric Response Routing Symmetric media RFC3581
- STUN: Session Traversal Utilities for NAT RFC5389
- Audio and Video: ICE, Interactive Connectivity Establishment RFC5245
- Instant Messaging and File Transfers: MSRP Relay Extension RFC4976
- MSRP Alternative Connection Model (ACM) RFC6135

Voice over IP

- RTP, A Transport Protocol for Real-Time Applications RFC3550
- Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP) RFC3605
- The Secure Real-time Transport Protocol (SRTP) RFC3711
- Generation and parsing of RFC 2833/telephone-events payload in both RTP and SDP RFC2833

Instant Messaging

- Common Presence and Instant Messaging (CPIM): RFC 3862
- Session Initiation Protocol (SIP) Extension for Instant Messaging RFC3428
- MSRP Protocol RFC4975
- Indication of Message Composition for Instant Messaging RFC3994
- Message Summary Event Package RFC3842
- File Transfer RFC5547

Desktop Sharing

- Variation of draft-garcia-mmusic-sdp-collaboration-00 using RFB over MSRP

Conferencing

- Conference Event Package RFC4575
- A Framework for Conferencing with the Session Initiation Protocol RFC4353
- SIP Call Control - Conferencing for User Agents RFC4579

    - 5.1 INVITE: Joining a Conference Using the Conference URI - Dial-In
    - 5.2 INVITE: Adding a Participant by the Focus - Dial-Out
    - 5.5 REFER: Requesting a Focus to Add a New Resource to a Conference
    - 5.11 REFER with BYE: Requesting a Focus to Remove a Participant from a Conference
- MSRP ad-hoc multi-party chat sessions draft-ietf-simple-chat-08

Presence

- SIP Specific Event Notification (SUBSCRIBE and NOTIFY methods) RFC3265
- SIP Extension for Event State Publication (PUBLISH method) RFC3903
- Presence Data Model (PIDF) RFC3863, RFC3379, RFC4479
- Watcher-info Event Package RFC3857, RFC3858
- Rich Presence Extensions to PIDF RFC4480
- Contact Information Extension to PIDF RFC4482
- User Agent Capability Extension to PIDF RFC5196
- XCAP Protocol RFC4825
- Common Policy RFC4745
- Presence Rules RFC5025
- Resource Lists RFC4826
- RLS Services RFC4826
- PIDF manipulation RFC4827
- XCAP Diff RFC5874
- OMA Reference Release Definition for XDM v1.1 and Presence SIMPLE v1.1 Implementation Guidelines
- OMA XML Document Management V1.1

Installation Guide

For an up to date installation guide visit http://sipsimpleclient.com

Prerequisites

A physical sound card is required.

The following computing platforms have been tested and are fully supported:

- Linux (i386 and amd64 architectures)
- MacOSX (Intel 32 bit architecture)
- Microsoft Windows (XP, Vista and 7)

The software can be theoretically ported to any operating systems supported by the cross-platform PortAudio audio library.

Manual Installation

Tar Archives

The software is available as a tar archive at:

```
http://download.ag-projects.com/SipClient/
```

Version Control Repository

The source code is managed using darcs version control tool. The darcs repository can be fetched with:

SIP SIMPLE client SDK

```
darcs get http://devel.ag-projects.com/repositories/python-sipsimple
```

Command Line Tools

```
darcs get http://devel.ag-projects.com/repositories/sipclients
```

To obtain the incremental changes after the initial get, go to the python-sipsimple and sipclients directory and run:

```
darcs pull –a
```

Up to date installation instructions are available in the folowing directory:

```
python-sipsimple/docs/
```

Install the software dependencies according to the instructions.

Then go into the directory where you have unpacked the software and run:

```
cd python-sipsimple
sudo python setup.py install
```

The software will be installed in the default Python site-packages directory on your system.

Debian Packages

Install the AG Projects debian software signing key:

```
wget http://download.ag-projects.com/agp-debian-gpg.key
sudo apt-key add agp-debian-gpg.key
```

Add these lines to /etc/apt/sources.list:

Debian Unstable (Sid)

```
deb   http://ag-projects.com/debian unstable main
deb-src http://ag-projects.com/debian unstable main
```

Debian Stable (Squeeze)

```
deb   http://ag-projects.com/debian stable main
deb-src http://ag-projects.com/debian stable main
```

Ubuntu Natty (11.04)

```
deb   http://ag-projects.com/ubuntu natty main
deb-src http://ag-projects.com/ubuntu natty main
```

Ubuntu Lucid (10.04)

```
deb   http://ag-projects.com/ubuntu lucid main
deb-src http://ag-projects.com/ubuntu lucid main
```

Ubuntu Maverick (10.10)

```
deb   http://ag-projects.com/ubuntu maverick main
deb-src http://ag-projects.com/ubuntu maverick main
```

Update the list of available packages:

```
sudo apt-get update
```

Install SIP SIMPLE client SDK

```
sudo apt-get install python-sipsimple
```

Install Command Line Tools

```
sudo apt-get install sipclients
```

MacOSX 10.6 Snow Leopard

The installation procedure consists of the steps described below:

- Step 1. Prerequisites
- Step 2. Install Dependencies
- Step 3. Install SIP SIMPLE Client SDK

Prerequisites

- MacOSX 10.6 (Snow Leopard)
- Intel 32bit architecture
- Apple Developer tools (XCode 4.2)
- darcs version control tool from http://www.darcs.net

The procedure below relies on the standard available Python interpreter that comes with MacOSX Snow Leopard (version 2.6) and Xcode version 4.2. Make sure that during the building process you are not involving external tools or libraries like the ones provided by Fink or Darwin Ports distributions.

Install Dependencies

python-gnutls	http://pypi.python.org/simple/python-gnutls	>=1.2.2
python-application	http://pypi.python.org/simple/python-application	>=1.2.8
Python-backports	http://download.ag-projects.com/SipClient	>=1.0.0
python-lxml	http://codespeak.net/lxml	>=2.1.2
python-eventlet	http://download.ag-projects.com/SipClient	=0.8.11.5
python-greenlet	http://download.ag-projects.com/SipClient	=0.4.0
python-cjson	http://pypi.python.org/pypi/python-cjson/	>=1.0.5
cython	http://www.cython.org	=0.14.1
dnspython	http://www.dnspython.org	>=1.6.0
twisted	http://twistedmatrix.com/trac	>=8.1.0
zope-interface	http://www.zope.org	>=3.3.1
python-imaging	http://pypi.python.org/pypi/PIL/	>=1.1.6
python-dateutil	http://niemeyer.net/python-dateutil	>=1.4

Build and install C dependencies, the software will be installed under /usr/local folder:

```
# Install GNUTLS dependencies
curl -O ftp://ftp.gnupg.org/gcrypt/libgpg-error/libgpg-error-
1.10.tar.bz2
tar -xjvf libgpg-error-1.10.tar.bz2
cd libgpg-error-1.10
make clean
CFLAGS="-arch i386" ./configure --prefix=/usr/local --disable-static --
disable-dependency-tracking
make
sudo make install
cd ..
```

```
curl -O http://ftp.gnu.org/pub/gnu/libtasn1/libtasn1-2.10.tar.gz
tar zxvf libtasn1-2.10.tar.gz
cd libtasn1-2.10
make clean
CFLAGS="-arch i386" ./configure --disable-dependency-tracking
make
sudo make install
cd ..

curl -O ftp://ftp.gnupg.org/gcrypt/libgcrypt/libgcrypt-1.5.0.tar.bz2
tar -xjvf libgcrypt-1.5.0.tar.bz2
cd libgcrypt-1.5.0
make clean
CFLAGS="-arch i386" ./configure --prefix=/usr/local --with-gpg-error-
prefix=/usr/local --disable-static --disable-dependency-tracking --
disable-asm
make
sudo make install
cd ..

# Install GNUTLS
curl -O http://ftp.gnu.org/pub/gnu/gnutls/gnutls-2.12.14.tar.bz2
tar -xjvf gnutls-2.12.14.tar.bz2
cd gnutls-2.12.14
make clean
CFLAGS="-arch i386" CXXFLAGS="-arch i386" ./configure --
prefix=/usr/local --with-libgcrypt-prefix=/usr/local --disable-static -
-disable-dependency-tracking --without-p11-kit --with-libgcrypt
make
sudo make install
cd ..
```

Build and install the Python dependencies by using the easy_install tool. The software will be installed in /Library/Python/2.6/site-packages folder.

You must become root first. The export the following environment variables before starting the build process:

```
sudo -s
export CC="gcc -isysroot /Developer/SDKs/MacOSX10.6.sdk"
export ARCHFLAGS="-arch i386"
export LDSHARED="gcc -Wl,-F. -bundle -undefined dynamic_lookup -
isysroot /Developer/SDKs/MacOSX10.6.sdk"

easy_install -U python-gnutls dnspython twisted python-application PIL
cython==0.14 python-dateutil pyOpenSSL

# Install lxml python module
STATIC_DEPS=true CFLAGS="-arch i386" easy_install lxml

# Stop being root
exit
```

Install SIP SIMPLE client SDK

The SDK consists of four parts:

- Eventlet and Greenlet
- XCAP library
- MSRP library
- SIP SIMPLE library

```
# Eventlet
if [ -d python-eventlet ]; then
        cd python-eventlet
        darcs pull -a
        sudo python setup.py install
else
        darcs get http://devel.ag-projects.com/repositories/python-
eventlet
        cd python-eventlet
        sudo python setup.py install
fi
cd ..

# Greenlet
if [ -d python-greenlet ]; then
        cd python-greenlet
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
greenlet
        cd python-greenlet
fi
sudo -s
export CC="gcc -isysroot /Developer/SDKs/MacOSX10.6.sdk"
export ARCHFLAGS="-arch i386"
export LDSHARED="gcc -Wl,-F. -bundle -undefined dynamic_lookup -
isysroot /Developer/SDKs/MacOSX10.6.sdk"
python setup.py build
python setup.py install
exit
cd ..

# Backports
if [ -d python-backports ]; then
        cd python-backports
        darcs pull -a
        sudo python setup.py install
else
        darcs get http://devel.ag-projects.com/repositories/python-
backports
        cd python-backports
        sudo python setup.py install
fi
cd ..

# XCAP library
if [ -d python-xcaplib ]; then
```

```
        cd python-xcaplib
        darcs pull -a
        sudo python setup.py install
else
        darcs get http://devel.ag-projects.com/repositories/python-
xcaplib
        cd python-xcaplib
        sudo python setup.py install
fi
cd ..

# MSRP library
if [ -d python-msrplib ]; then
        cd python-msrplib
        darcs pull -a
        sudo python setup.py install
else
        darcs get http://devel.ag-projects.com/repositories/python-
msrplib
        cd python-msrplib
        sudo python setup.py install
fi
cd ..
```

Note: 64 bit architecture is not yet fully supported, namely there is hissing sound in the audio input layer that manifests itself only when building in 64 bit mode. Until a fix is found, the workaround is to use the 32 bit mode.

Build and install SIP SIMPLE library:

```
sudo -s
# build only for 32 bit architechture to avoid the audio input bug
export SIPSIMPLE_OSX_ARCH="i386"
python setup.py build_ext --pjsip-clean-compile
python setup.py install

Additional, you can install the command line interface scripts that can
be
used to test the SDK capabilities.

if [ -d sipclients ]; then
    cd sipclients
    darcs pull -a
else
    darcs get http://devel.ag-projects.com/repositories/sipclients
    cd sipclients
fi
sudo python setup.py install
cd ..
```

Additional, you can install the command line interface scripts that can be used to test the SDK.

```
if [ -d sipclients ]; then
```

```
      cd sipclients
      darcs pull -a
else
      darcs get http://devel.ag-projects.com/repositories/sipclients
fi
cd..

sudo python setup.py install
```

To use the sipclients command line tools, you must force the system Python interpreter to use the 32 bit mode as by default the Python interpreter uses the 64 mode while the SIP SIMPLE core is built for 32 bits:

```
export VERSIONER_PYTHON_PREFER_32_BIT=yes
```

MacOSX 10.7 Lion

The installation procedure consists of the steps described below:

- Step 1. Prerequisites
- Step 2. Install Dependencies
- Step 3. Install SIP SIMPLE Client SDK

Prerequisites

- MacOSX 10.7 (Lion)
- Intel 32bit architecture
- Apple Developer tools (XCode 4.2)
- darcs version control tool from http://www.darcs.net

The procedure below relies on Python interpreter that comes with MacOSX Lion (version 2.6) and Xcode version 4.2. Make sure that during the building process you are not involving external tools or libraries like the ones provided by Fink or Darwin Ports distributions.

Install Dependencies

python-gnutls	http://pypi.python.org/simple/python-gnutls	>=1.2.2
python-application	http://pypi.python.org/simple/python-application	>=1.2.8
Python-backports	http://download.ag-projects.com/SipClient	>=1.0.0
python-lxml	http://codespeak.net/lxml	>=2.1.2
python-eventlet	http://download.ag-projects.com/SipClient	=0.8.11.5
python-greenlet	http://download.ag-projects.com/SipClient	=0.4.0
python-cjson	http://pypi.python.org/pypi/python-cjson/	>=1.0.5
cython	http://www.cython.org	=0.14.1
dnspython	http://www.dnspython.org	>=1.6.0
twisted	http://twistedmatrix.com/trac	>=8.1.0
zope-interface	http://www.zope.org	>=3.3.1
python-imaging	http://pypi.python.org/pypi/PIL/	>=1.1.6
python-dateutil	http://niemeyer.net/python-dateutil	>=1.4

Build and install C dependencies, the software will be installed under /usr/local folder:

```
# Install GNUTLS dependencies
curl -O ftp://ftp.gnupg.org/gcrypt/libgpg-error/libgpg-error-
1.10.tar.bz2
tar -xjvf libgpg-error-1.10.tar.bz2
cd libgpg-error-1.10
make clean
CFLAGS="-arch i386" ./configure --prefix=/usr/local --disable-static --
disable-dependency-tracking
make
sudo make install
cd ..
```

```
curl -O http://ftp.gnu.org/pub/gnu/libtasn1/libtasn1-2.10.tar.gz
tar zxvf libtasn1-2.10.tar.gz
cd libtasn1-2.10
make clean
CFLAGS="-arch i386" ./configure --disable-dependency-tracking
make
sudo make install
cd ..

curl -O ftp://ftp.gnupg.org/gcrypt/libgcrypt/libgcrypt-1.5.0.tar.bz2
tar -xjvf libgcrypt-1.5.0.tar.bz2
cd libgcrypt-1.5.0
make clean
CFLAGS="-arch i386" ./configure --prefix=/usr/local --with-gpg-error-
prefix=/usr/local --disable-static --disable-dependency-tracking --
disable-asm
make
sudo make install
cd ..

# Install GNUTLS
curl -O http://ftp.gnu.org/pub/gnu/gnutls/gnutls-2.12.14.tar.bz2
tar -xjvf gnutls-2.12.14.tar.bz2
cd gnutls-2.12.14
make clean
CFLAGS="-arch i386" CXXFLAGS="-arch i386" ./configure --
prefix=/usr/local --with-libgcrypt-prefix=/usr/local --disable-static -
-disable-dependency-tracking --without-p11-kit --with-libgcrypt
make
sudo make install
cd ..
```

Build and install the Python dependencies by using the easy_install tool. The software will be installed in /Library/Python/2.6/site-packages folder.

You must become root first. The export the following environment variables before starting the build process:

```
sudo -s
export CC="gcc -isysroot /Developer/SDKs/MacOSX10.6.sdk"
export ARCHFLAGS="-arch i386"
export LDSHARED="gcc -Wl,-F. -bundle -undefined dynamic_lookup -
isysroot /Developer/SDKs/MacOSX10.6.sdk"

easy_install -U python-gnutls dnspython twisted python-application PIL
cython==0.14 python-dateutil pyOpenSSL

# Install lxml python module
STATIC_DEPS=true CFLAGS="-arch i386" easy_install lxml

# Stop being root
exit
```

Install SIP SIMPLE client SDK

The SDK consists of four parts:

- Eventlet and Greenlet
- XCAP library
- MSRP library
- SIP SIMPLE library

```
# Eventlet
if [ -d python-eventlet ]; then
        cd python-eventlet
        darcs pull -a
        sudo python setup.py install
else
        darcs get http://devel.ag-projects.com/repositories/python-
eventlet
        cd python-eventlet
        sudo python setup.py install
fi
cd ..

# Greenlet
if [ -d python-greenlet ]; then
        cd python-greenlet
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
greenlet
        cd python-greenlet
fi
sudo -s
export CC="gcc -isysroot /Developer/SDKs/MacOSX10.6.sdk"
export ARCHFLAGS="-arch i386"
export LDSHARED="gcc -Wl,-F. -bundle -undefined dynamic_lookup -
isysroot /Developer/SDKs/MacOSX10.6.sdk"
python setup.py build
python setup.py install
exit
cd ..

# Backports
if [ -d python-backports ]; then
        cd python-backports
        darcs pull -a
        sudo python setup.py install
else
        darcs get http://devel.ag-projects.com/repositories/python-
backports
        cd python-backports
        sudo python setup.py install
fi
cd ..

# XCAP library
if [ -d python-xcaplib ]; then
        cd python-xcaplib
```

```
        darcs pull -a
        sudo python setup.py install
else
        darcs get http://devel.ag-projects.com/repositories/python-
xcaplib
        cd python-xcaplib
        sudo python setup.py install
fi
cd ..

# MSRP library
if [ -d python-msrplib ]; then
        cd python-msrplib
        darcs pull -a
        sudo python setup.py install
else
        darcs get http://devel.ag-projects.com/repositories/python-
msrplib
        cd python-msrplib
        sudo python setup.py install
fi
cd ..
```

Note: 64 bit architecture is not yet fully supported, namely there is hissing sound in the audio input layer that manifests itself only when building in 64 bit mode. Until a fix is found, the workaround is to use the 32 bit mode.

Build and install SIP SIMPLE library:

```
sudo -s
# build only for 32 bit architechture to avoid the audio input bug
export SIPSIMPLE_OSX_ARCH="i386"
python setup.py build_ext --pjsip-clean-compile
python setup.py install

Additional, you can install the command line interface scripts that can
be
used to test the SDK capabilities.

if [ -d sipclients ]; then
    cd sipclients
    darcs pull -a
else
    darcs get http://devel.ag-projects.com/repositories/sipclients
    cd sipclients
fi
sudo python setup.py install
cd ..
```

Additional, you can install the command line interface scripts that can be used to test the SDK.

```
if [ -d sipclients ]; then
```

```
      cd sipclients
      darcs pull -a
else
      darcs get http://devel.ag-projects.com/repositories/sipclients
fi
cd..

sudo python setup.py install
```

To use the sipclients command line tools, you must force the system Python interpreter to use the 32 bit mode as by default the Python interpreter uses the 64 mode while the SIP SIMPLE core is built for 32 bits:

```
export VERSIONER_PYTHON_PREFER_32_
```

Windows Installer

Prerequisites

The building process is designed to work with MinGW compiler. A proper MSYS/MinGW setup is necessary along with Python (>= 2.5). The development version of the following packages are also needed:

- openssl
- gnutls >=2.4.1
- python-setuptools =>0.6c9
- subversion and darcs version control tools

Required MSYS and MinGW packages:

MSYS (http://sourceforge.net/projects/mingw/files/MSYS/)

- MsysCORE (bin)
- libregex (dll)
- libtermcap (dll)
- coreutils (bin)
- libintl (dll)
- libiconv (dll)
- bash (bin)
- wget (bin)
- make (bin)
- sed (bin)
- grep (bin)
- gawk (bin)
- findutils (bin)
- patch (bin)
- tar (bin)
- bzip2 (bin)
- gzip (bin)
- diffutils (bin)

MinGW (http://sourceforge.net/projects/mingw/files/MinGW/)

- gcc-core (bin)
- libgcc (dll)
- gcc-c++ (bin)
- binutils (bin)
- less (bin)
- gmp (dev)
- libgmp (dll)
- pthreads (dev)
- libpthread (dll)
- mpc (dev)
- libmpc (dll)
- mpfr (dev)
- libmpfr (dll)
- mingwrt (dev)
- libz (dll)

- gdb (bin)
- libexpat (dll)
- win32api (dev)

To install the above dependencies in an easy way, download AG Projects installer from:

http://download.ag-projects.com/SipClient/Windows/SipSimpleIDE.exe

The installer includes the following components:

- MSYS environment
- MinGW compiler
- Darcs and subversion version control tools
- Python (2.6.5)
- python-srtuptools (0.6c11)
- GNUTLS (2.8.6)
- OpenSSL (1.0.0a)
- Apple Bonjour SDK
- Miscrosoft VC 2008 Redistributable

When using the installer select the default options proposed by the installer.

After the above dependencies have been installed, the distutils Python package needs to be configured to use MinGW as the compiler. Create the file

C:\Python26\Lib\distutils\distutils.cfg with the following content:

```
-- BEGIN distutils.cfg --
[build]
compiler=mingw32
[build_ext]
compiler=mingw32
-- END distutils.cfg --
```

Install python dependencies

The following python packages need to be installed. Notice the minimum version numbers:

python-gnutls	http://pypi.python.org/simple/python-gnutls	dev
python-application	http://pypi.python.org/simple/python-application	dev
python-lxml	http://codespeak.net/lxml	==2.2.4
python-eventlet	http://download.ag-projects.com/SipClient	=0.8.11.4
python-greenlet	http://download.ag-projects.com/SipClient	=0.4.0
python-cjson	http://pypi.python.org/pypi/python-cjson/	>=1.0.5
cython	http://www.cython.org	=0.12.1
dnspython	http://www.dnspython.org	>=1.6.0
twisted	http://twistedmatrix.com/trac	>=8.1.0
zope-interface	http://www.zope.org	>=3.3.1

You must use the easy_install script provided by the python-setuptools package to install the packages:

```
easy_install -U cython==0.14.1 dnspython twisted lxml==2.2.4
```

Some packages need to be installed manually:

```
# python-application
if [ -d python-application ]; then
        cd python-application
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
application
        cd python-application
fi
python setup.py install
cd ..

# python-cjson
if [ -d python-cjson ]; then
        cd python-cjson
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
cjson
        cd python-cjson
fi
python setup.py build
python setup.py install
cd ..

# python-gnutls
if [ -d python-gnutls ]; then
        cd python-gnutls
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
gnutls
        cd python-gnutls
fi
python setup.py build
python setup.py install
cd ..
```

Install SIP SIMPLE client SDK

The SDK consists of four parts:

1. Eventlet and Greenlet
2. XCAP library
3. MSRP library
4. SIP SIMPLE library

```
# Greenlet
if [ -d python-greenlet ]; then
        cd python-greenlet
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
greenlet
        cd python-greenlet
fi
python setup.py install
cd ..

# Eventlet
if [ -d python-eventlet ]; then
        cd python-eventlet
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
eventlet
        cd python-eventlet
fi
python setup.py install
cd ..

# XCAP library
if [ -d python-xcaplib ]; then
        cd python-xcaplib
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
xcaplib
        cd python-xcaplib
fi
python setup.py install
cd ..

# MSRP library
if [ -d python-msrplib ]; then
        cd python-msrplib
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
msrplib
        cd python-msrplib
fi
python setup.py install
cd ..

# SIP SIMPLE
if [ -d python-sipsimple ]; then
        cd python-sipsimple
        darcs pull -a
else
        darcs get http://devel.ag-projects.com/repositories/python-
sipsimple
        cd python-sipsimple
fi
python setup.py build_ext --pjsip-clean-compile
```

```
python setup.py install
cd ..
```

The software has been installed in C:\Python26\Lib\site-packages

Testing Guide

To test SIP SIMPLE client SDK features, you can use the Command Line Tools provided by the **sipclients** package.

SIP Account

By default the Bonjour account is enabled and set as default. To use the Command Line Tools on the public Internet, you must setup at least a SIP account.

You can register a SIP account for free at http://sip2sip.info.

```
sip-settings -a add user@domain pasword
sip-settings -a default user@domain
```

sip-settings

```
Implemented in sipclients/sip-settings
```

Manages global and SIP account settings used by middleware and Command Line Tools.

```
adigeo@ag-imac3:~$sip-settings
Usage: sip-settings [--general|--account] [options] command [arguments]
       sip-settings --general show
       sip-settings --general set key1=value1 [key2=value2 ...]
       sip-settings --account list
       sip-settings --account add user@domain password
       sip-settings --account delete user@domain|ALL
       sip-settings --account show [user@domain|ALL]
       sip-settings --account set [user@domain|ALL] key1=value1|DEFAULT
       sip-settings --account default user@domain

This script is used to manage the SIP SIMPLE middleware settings.

Options:
  -h, --help            show this help message and exit
  -c FILE, --config-file=FILE
                        The path to a configuration file to use. This
                        overrides the default location of the
configuration
                        file.
  -a, --account         Manage SIP accounts' settings
  -g, --general         Manage general SIP SIMPLE middleware settings
To use the command line tools you must create at least one SIP account:
sip-settings --account add user@domain password
```

sip-register

```
Implemented in sipclients/sip-register
```

You can use this script to Register a SIP end-point with a SIP Registrar or broadcast the local SIP URI using Bonjour mDNS.

Description

SIP protocol offers a discovery capability. If a user wants to initiate a session with another user, he must discover the current host(s) at which the destination user is reachable. To do this, SIP network elements consult an abstract service known as a location service, which provides address bindings for a particular domain. Registration entails sending a REGISTER request to a special type of UAS known as a registrar. A registrar acts as the front end to the location service for a domain, reading and writing mappings based on the contents of REGISTER requests. This location service is then typically consulted by a proxy server that is responsible for routing requests for that domain.

This script implements REGISTER method, which registers the contact (ip, port and transport) for a given address of record (SIP address).

```
adigeo@ag-imac3:~$sip-register -h
Usage: sip-register [options]

This script will register a SIP account to a SIP registrar and refresh
it
while the program is running. When Ctrl+D is pressed it will
unregister.

Options:
  -h, --help            show this help message and exit
  -a ACCOUNT_NAME, --account-name=ACCOUNT_NAME
                        The account name from which to read account
settings.
                        Corresponds to section Account_NAME in the
                        configuration file.
  -s, --trace-sip       Dump the raw contents of incoming and outgoing
SIP
                        messages (disabled by default).
  -j, --trace-pjsip     Print PJSIP logging output (disabled by
default).
  -r MAX_REGISTERS, --max-registers=MAX_REGISTERS
                        Max number of REGISTERs sent (default 1, set to
0 for
                        infinite).
```

Example

```
adigeo@ag-imac3:~$sip-register
Using account 31208005169@ag-projects.com
Registration succeeded at 85.17.186.7:5060;transport=udp.
```

```
Contact: sip:xqdwrctb@192.168.1.6:58481 (expires in 600 seconds).
Other registered contacts:
  sip:31208005169@192.168.1.123:5060 (expires in 262 seconds)
  sip:31208005169@192.168.1.122:5062;line=634g6j67 (expires in 360
seconds)
  sip:31208005169@192.168.1.1;uniq=5B2860C44383A3D6705629A7E1FB8
(expires in 734 seconds)
Registration ended: 200 OK.
```

sip-audio-session

```
Implemented in sipclients/sip-audio-session
```

Setup a single SIP audio session using RTP/sRTP media.

Description

This script can be used for interactive audio session or for scripting alarms. The script returns appropriate shell response codes for failed or successful sessions. The script can be setup to auto answer and auto hangup after predefined number of seconds, detects SIP negative response codes, missing ACK and the lack of RTP media after a session has been established. Once the media stream is connected, the outcome of the ICE negotiation and the selected RTP candidates are displayed.

```
adigeo@ag-blink:~$sip-audio-session -h
Usage: sip-audio-session [options] [user@domain]

This script can sit idle waiting for an incoming audio session, or
initiate an
outgoing audio session to a SIP address. The program will close the
session
and quit when Ctrl+D is pressed.

Options:
  -h, --help            show this help message and exit
  -a NAME, --account=NAME
                        The account name to use for any outgoing
traffic. If
                        not supplied, the default account will be used.
  -c FILE, --config-file=FILE
                        The path to a configuration file to use. This
                        overrides the default location of the
configuration
                        file.
  -s, --trace-sip       Dump the raw contents of incoming and outgoing
SIP
                        messages.
  -j, --trace-pjsip     Print PJSIP logging output.
  -n, --trace-notifications
                        Print all notifications (disabled by default).
  -S, --disable-sound   Disables initializing the sound card.
  --auto-answer         Interval after which to answer an incoming
session
                        (disabled by default). If the option is
specified but
                        the interval is not, it defaults to 0 (accept
the
                        session as soon as it starts ringing).
  --auto-hangup         Interval after which to hang up an established
session
                        (disabled by default). If the option is
specified but
```

```
                          the interval is not, it defaults to 0 (hangup
the
                          session as soon as it connects).
  -b, --batch             Run the program in batch mode: reading input
from the
                          console is disabled and the option --auto-
answer is
                          implied. This is particularly useful when
running this
                          script in a non-interactive environment.
  -D, --daemonize         Enable running this program as a deamon. This
option
                          implies --disable-sound, --auto-answer and --
batch.
```

Incoming Session

```
adigeo@ag-blink:~$sip-audio-session
Using account 31208005169@ag-projects.com
Logging SIP trace to file "/Users/adigeo/Library/Application
Support/Blink/logs/sip_trace.txt"
Logging PJSIP trace to file "/Users/adigeo/Library/Application
Support/Blink/logs/pjsip_trace.txt"
Available audio input devices: None, system_default, Built-in Input,
Built-in Microphone
Available audio output devices: None, system_default, Built-in Output
Using audio input device: Built-in Microphone
Using audio output device: Built-in Output
Using audio alert device: Built-in Output

Available control keys:
  s: toggle SIP trace on the console
  j: toggle PJSIP trace on the console
  n: toggle notifications trace on the console
  p: toggle printing RTP statistics on the console
  h: hang-up the active session
  r: toggle audio recording
  m: mute the microphone
  i: change audio input device
  o: change audio output device
  a: change audio alert device
  <>: adjust echo cancellation
  SPACE: hold/unhold
  Ctrl-d: quit the program
  ?: display this help message

2009-08-25 16:37:12 Registered contact
"sip:hxsyungk@192.168.1.124:59164" for sip:31208005169@ag-projects.com
at 81.23.228.150:5060;transport=udp (expires in 600 seconds).
Other registered contacts:
  sip:31208005169@192.168.1.123:5060 (expires in 274 seconds)
  sip:kwbfxyvl@192.168.1.124:59116 (expires in 522 seconds)
  sip:ilmegvkp@192.168.1.124:59003 (expires in 339 seconds)
  sip:31208005169@192.168.1.1;uniq=5B2860C44383A3D6705629A7E1FB8
(expires in 1162 seconds)
Detected NAT type: Port Restricted
```

```
Incoming audio session from 'sip:adi@umts.ro', do you want to accept?
(y/n)
Audio session established using "speex" codec at 16000Hz
Audio RTP endpoints 192.168.1.124:50378 <-> 85.17.186.6:58868
RTP audio stream is encrypted
Remote SIP User Agent is "Blink-0.9.0"
Remote party has put the audio session on hold
Audio session is put on hold
Audio session ended by remote party
Session duration was 6 seconds
2009-08-25 16:37:44 Registration ended.
```

Outgoing Session

```
adigeo@ag-blink:~$sip-audio-session -a umts ag@ag-projects.com
Using account adi@umts.ro
Logging SIP trace to file "/Users/adigeo/Library/Application
Support/Blink/logs/sip_trace.txt"
Logging PJSIP trace to file "/Users/adigeo/Library/Application
Support/Blink/logs/pjsip_trace.txt"
Available audio input devices: None, system_default, Built-in Input,
Built-in Microphone
Available audio output devices: None, system_default, Built-in Output
Using audio input device: Built-in Microphone
Using audio output device: Built-in Output
Using audio alert device: Built-in Output

Available control keys:
  s: toggle SIP trace on the console
  j: toggle PJSIP trace on the console
  n: toggle notifications trace on the console
  p: toggle printing RTP statistics on the console
  h: hang-up the active session
  r: toggle audio recording
  m: mute the microphone
  i: change audio input device
  o: change audio output device
  a: change audio alert device
  <>: adjust echo cancellation
  SPACE: hold/unhold
  Ctrl-d: quit the program
  ?: display this help message

Initiating SIP audio session from 'sip:adi@umts.ro' to 'sip:ag@ag-
projects.com' via sip:85.17.186.7:5060;transport=udp...
Audio session established using "speex" codec at 16000Hz
ICE negotiation succeeded in 1s:412
Audio RTP endpoints 192.168.1.124:50852 (ICE type host) <->
192.168.1.124:50871 (ICE type host)
RTP audio stream is encrypted
Audio session is put on hold
Remote party has put the audio session on hold
Detected NAT type: Port Restricted
Ending audio session...
Audio session ended by local party
```

```
Session duration was 7 seconds
```

Alarm System

sip-audio-session script can be used for end-to-end testing of a SIP service including the RTP media path. The following failures can be detected:

1. Timeout
2. Negative response code
3. Lack of RTP media after the SIP session has been established
4. Missing ACK

To setup the alarm system start periodically a caller script from a monitoring software using the following arguments:

```
sip-audio-session --auto-hangup user@domain
```

Where the user@domain has been configured as the SIP account of the listener, can be an answering machine on the PSTN network. The caller script hangs up after each call. The shell return code can be used to determine if the session setup has failed.

To receive calls and answer them automatically you can also use sip_audio_session script as follows:

```
sip-audio-session --daemonize
```

You must run the script as user root. The --daemonize option puts the client in the background and the logging goes to /var/log/syslog. The program saves its pid file to /var/run/sip_audio_session.pid.

sip-session

```
Implemented in sipclients/sip-session
```

Setup one or more SIP sessions with Audio (RTP/sRTP), IM and File Transfer (MSRP).

Description

sip-session command line script is a show-case for the powerful features of SIP SIMPLE development kit related to establishing, modifying and terminating SIP sessions with multiple media types like VoIP, Instant Messaging and File Transfer.

The script has the following features:

1. Registers with a SIP registrar and is available for incoming sessions
2. Switches between multiple sessions and provides in-call controls like Hold and Mute
3. Handles outgoing SIP sessions with combinations of media types based on RTP and MSRP protocols
4. Performs NAT traversal using ICE and MSRP relay extension
5. Provides control for the input, output and alert audio devices
6. Records the RTP audio streams (input, output or combined)
7. Enable text input and output for Instant Messaging sessions
8. Provides File Transfer capability with progress indicator
9. Gives access to real-time traces of involved protocols (DNS, SIP and MSRP)

Example

```
adigeo@ag-imac3:~$sip-session
Using account adi@umts.ro
adi@umts.ro> /help
General commands:
 /call {user[@domain]}: call the specified user using audio and chat
 /audio {user[@domain]} [+chat]: call the specified user using audio
and possibly chat
 /chat {user[@domain]} [+audio]: call the specified user using chat and
possibly audio
 /send {user[@domain]} {file}: initiate a file transfer with the
specified user
 /next: select the next connected session
 /prev: select the previous connected session
 /sessions: show the list of connected sessions
 /trace [[+|-]sip] [[+|-]msrp] [[+|-]pjsip] [[+|-]notifications]:
toggle/set tracing on the console (ctrl-x s | ctrl-x m | ctrl-x j |
ctrl-x n)
 /rtp [on|off]: toggle/set printing RTP statistics and ICE negotiation
results on the console (ctrl-x p)
 /mute [on|off]: mute the microphone (ctrl-x u)
 /input [device]: change audio input device (ctrl-x i)
 /output [device]: change audio output device (ctrl-x o)
 /alert [device]: change audio alert device (ctrl-x a)
 /echo [+|-][value]: adjust echo cancellation (ctrl-x < | ctrl-x >)
 /quit: quit the program (ctrl-x q)
```

```
 /help: display this help message (ctrl-x ?)

In call commands:
 /hangup: hang-up the active session (ctrl-x h)
 /dtmf {0-9|*|#|A-D}...: send DTMF tones (ctrl-x 0-9|*|#|A-D)
 /record [on|off]: toggle/set audio recording (ctrl-x r)
 /hold [on|off]: hold/unhold (ctrl-x SPACE)
 /add {chat|audio}: add a stream to the current session
 /remove {chat|audio}: remove a stream from the current session
 /add_participant {user@domain}: add the specified user to the
conference
 /remove_participant {user@domain}: remove the specified user from the
conference
 /transfer {user@domain}: transfer (using blind transfer) callee to the
specified destination

Available audio input devices: None, system_default, Built-in Input,
Built-in Microphone
Available audio output devices: None, system_default, Built-in Output
Using audio input device: Built-in Microphone (system default device)
Using audio output device: Built-in Output (system default device)
Using audio alert device: Built-in Output
Type /help to see a list of available commands.
2009-10-29 22:42:14 Registered contact
"sip:puioxbqy@192.168.1.124:50150" (expires in 600 seconds).
Other registered contacts:
  sip:jiozqyud@192.168.1.124:49569 (expires in 423 seconds)
Detected NAT type: Port Restricted
adi@umts.ro>
```

ICE connectivity checks results:

```
adi@umts.ro> /rtp
Output of RTP statistics and ICE negotiation results on console is now
activated
adi@umts.ro> /audio ag@sip2sip.info
Initiating SIP session from 'sip:adi@umts.ro' to 'sip:ag@sip2sip.info'
via sip:81.23.228.150:5060;transport=udp...

ICE negotiation succeeded in 0s:644

Local ICE candidates:
(RTP)  95.97.50.27:55656             type srflx
(RTP)  192.168.1.122:55656           type host
(RTP)  10.211.55.2:55656             type host
(RTP)  10.37.129.2:55656             type host
(RTCP)       95.97.50.27:55890             type srflx
(RTCP)       192.168.1.122:55890           type host
(RTCP)       10.211.55.2:55890             type host
(RTCP)       10.37.129.2:55890             type host
(RTP)  81.23.228.150:51782           type prflx
(RTCP)       81.23.228.150:51783           type prflx

Remote ICE candidates:
(RTP)  81.23.228.150:51780           type relay
(RTCP)       81.23.228.150:51781           type relay
(RTP)  95.97.50.27:55876             type srflx
```

```
(RTP)  192.168.1.122:55876           type host
(RTP)  10.211.55.2:55876             type host
(RTP)  10.37.129.2:55876             type host
(RTCP)       95.97.50.27:54037             type srflx
(RTCP)       192.168.1.122:54037           type host
(RTCP)       10.211.55.2:54037             type host
(RTCP)       10.37.129.2:54037             type host

ICE connectivity check results:
(RTP)  192.168.1.122:55656 <--> 192.168.1.122:55876   Succeeded
(RTP)  10.211.55.2:55656 <--> 10.211.55.2:55876       Succeeded
(RTP)  10.37.129.2:55656 <--> 10.37.129.2:55876       Succeeded
(RTCP)       192.168.1.122:55890 <--> 192.168.1.122:54037   Succeeded
(RTCP)       10.211.55.2:55890 <--> 10.211.55.2:54037       Succeeded
(RTCP)       10.37.129.2:55890 <--> 10.37.129.2:54037       Succeeded
(RTP)  95.97.50.27:55656 <--> 95.97.50.27:55876       Succeeded
(RTCP)       95.97.50.27:55890 <--> 95.97.50.27:54037       Succeeded
(RTP)  81.23.228.150:51782 <--> 81.23.228.150:51780   Succeeded
(RTCP)       81.23.228.150:51783 <--> 81.23.228.150:51781   Succeeded

Audio session established using "G722" codec at 16000Hz
Audio RTP endpoints 192.168.1.122:55656 (ICE type host) <->
192.168.1.122:55876 (ICE type host)
```

sip-message

```
Implemented in sipclients/sip-message
```

Send and receive short messages in paging mode using SIP MESSAGE method.

Description

```
Usage: sip-message [options] [user@domain]

This will either sit idle waiting for an incoming MESSAGE request, or
send a
MESSAGE request to the specified SIP target. In outgoing mode the
program will
read the contents of the messages to be sent from standard input,
Ctrl+D
signalling EOF as usual. In listen mode the program will quit when
Ctrl+D is
pressed.

Options:
  -h, --help            show this help message and exit
  -a NAME, --account=NAME
                        The account name to use for any outgoing
traffic. If
                        not supplied, the default account will be used.
  -c FILE, --config-file=FILE
                        The path to a configuration file to use. This
                        overrides the default location of the
configuration
                        file.
  -s, --trace-sip       Dump the raw contents of incoming and outgoing
SIP
                        messages.
  -j, --trace-pjsip     Print PJSIP logging output.
  -n, --trace-notifications
                        Print all notifications (disabled by default).
  -b, --batch           Run the program in batch mode: reading control
input
                        from the console is disabled. This is
particularly
                        useful when running this script in a non-
interactive
                        environment.
  -m MESSAGE, --message=MESSAGE
                        Contents of the message to send. This disables
reading
                        the message from standard input.
```

Example for receiving a message

```
adigeo@ag-imac3:~$sip-message
Accounts available: 'alice', 'ew', 'mrg', 'pbx', 'tf', 'umts', 'unet',
default
Using default account: 31208005169@ag-projects.com
Registering ""Adrian G." <sip:31208005169@ag-projects.com>" at
81.23.228.150:5060
REGISTER was successful
Contact: <sip:4f855cb09b@192.168.1.6:51408> (expires in 300 seconds)
Other registered contacts:
<sip:31208005169@192.168.1.122:5062;line=634g6j67> (expires in 480
seconds)
<sip:5dac4295e9@192.168.1.6:51375> (expires in 95 seconds)
<sip:31208005169@192.168.1.123:5060> (expires in 77 seconds)
<sip:31208005169@192.168.1.1;uniq=5B2860C44383A3D6705629A7E1FB8>
(expires in 1563 seconds)
<sip:31208005169@80.101.96.20:61578> (expires in 3069 seconds)
Press Ctrl+D to stop the program.
Received MESSAGE from ""Adi UMTS" <sip:adi@umts.ro>", Content-Type:
text/plain
dsgsgddsgs
Received MESSAGE from ""Adi UMTS" <sip:adi@umts.ro>", Content-Type:
text/plain
Testing short text messages in page mode!
```

Example for sending a message

```
adigeo@ag-imac3:~$sip-message  -a umts ag@ag-projects.com
Accounts available: 'alice', 'ew', 'mrg', 'pbx', 'tf', 'umts', 'unet',
default
Using account 'umts': adi@umts.ro
Press Ctrl+D on an empty line to end input and send the MESSAGE
request.
dsgsgddsgs
Sending MESSAGE from ""Adi UMTS" <sip:adi@umts.ro>" to "<sip:ag@ag-
projects.com>" using proxy 81.23.228.150:5060
MESSAGE was accepted by remote party.
adigeo@ag-imac3:~$sip_message  -a umts ag@ag-projects.com
Accounts available: 'alice', 'ew', 'mrg', 'pbx', 'tf', 'umts', 'unet',
default
Using account 'umts': adi@umts.ro
Press Ctrl+D on an empty line to end input and send the MESSAGE
request.
Testing short text messages in page mode!
Sending MESSAGE from ""Adi UMTS" <sip:adi@umts.ro>" to "<sip:ag@ag-
projects.com>" using proxy 81.23.228.150:5060
MESSAGE was accepted by remote party.
```

Presence

You can use these scripts to Publish, Subscribe and handle incoming Notifies to and from a Presence Agent or manage documents on an XCAP server.

sip-publish-presence

PUBLISH presence to a Presence Agent

sip-subscribe-winfo

SUBSCRIBE to the watcher list for given SIP address on the Presence Agent

sip-subscribe-presence

SUBSCRIBE to Presence Event for a given SIP address

sip-subscribe-rls

SUBSCRIBE for Presence Event to a list managed by a Resource List Server

sip-subscribe-mwi

SUBSCRIBE for Message Waiting Indicator

xcap-directory

Show the XCAP documents stored in the XCAP server for the given account

xcap-icon

Stores and retrieves the icon for the given account

xcap-pres-rules

Manage the content of the pres-rules XCAP document

xcap-dialog-rules

Manage the content of the dialog-rules XCAP document

xcap-rls-services

Manage the content of a RLS services XCAP document

Developer Guide

The goal of SIP SIMPLE client SDK is to allow easy development of Real Time Applications based on SIP and related protocols. By using this SDK you can add Audio, Video, Instant Messaging, File Transfer and Desktop Sharing capabilities to an existing product or create a new product from scratch.

Prerequisites

To use SIP SIMPLE client SDK you must:

- Be familiar with Python programming language
- Have basic knowledge of SIP protocol
- Use a supported platform as described in the Installation Instructions

Middleware API

To develop your SIP application you use the Middleware API that hides the complexity of the interactions of the low level SIP, DNS, SDP, RTP, ICE, MSRP and XCAP protocols.

Low Level API

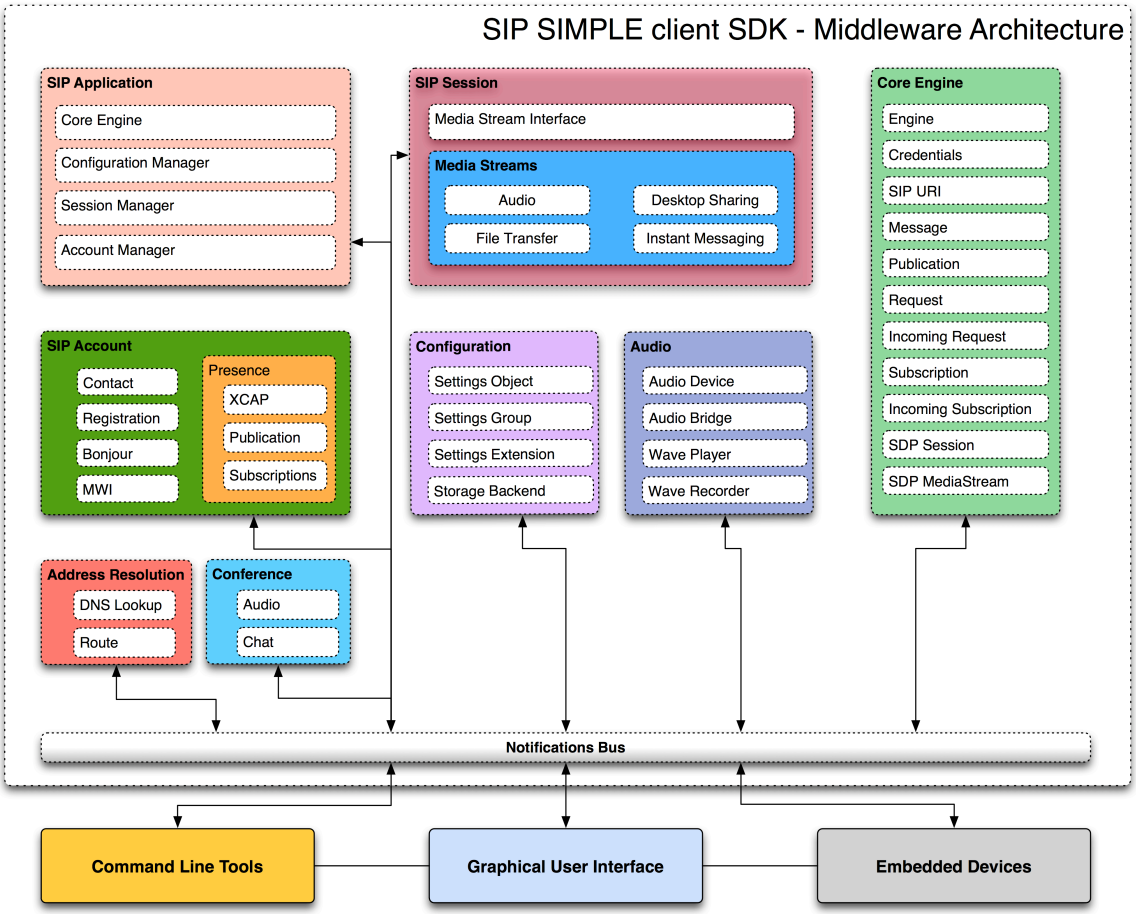
The following APIs provide granular control over their respective components:

- SIP Core API - SIP, RTP, ICE and Audio Engine
- MSRP API - Message Session Relay Protocol (MSRP) and its Relay Extension
- XCAP API - Manage presence policy documents on XCAP servers
- Payloads API - Payloads used for publication, subscription and notifications of SIP events

Middleware API

This chapter describes the Middleware API for SIP SIMPLE client SDK that can be used for developing a user interface (e.g. Graphical User Interface). The Middleware provides a non-blocking API that communicates with the user interface asynchronously by using Notifications.

For its configuration, the Middleware uses the Configuration API.



SIPApplication

```
Implemented in sipsimple/application.py
```

Implements a high-level application responsable for starting and stopping various sub-systems required to implement a fully featured SIP User Agent application. The SIPApplication class is a Singleton and can be instantiated from any part of the code, obtaining a reference to the same object. The SIPApplication takes care of initializing the following components:

1. the twisted thread
2. the configuration system, via the ConfigurationManager
3. the core Engine using the settings in the configuration
4. the AccountManager, using the accounts in the configuration
5. the SessionManager, in order to handle incoming sessions
6. two AudioBridges, using the settings in the configuration

The attributes in this class can be set and accessed on both this class and its subclasses, as they are implemented using descriptors which keep single value for each attribute, irrespective of the class from which that attribute is set/accessed. Usually, all attributes should be considered read-only.

methods

```
__init__(self)
```

Instantiates a new SIPApplication.

```
start(self, storage)
```

Starts the SIPApplication which initializes all the components in the correct order. The storage is saved as an attribute which other entities like the Configuration Manager will use to take the appropriate backend. If any error occurs with loading the configuration, the exception raised by the ConfigurationManager is propagated by this method and SIPApplication can be started again. After this, any fatal errors will result in the SIPApplication being stopped and unusable, which means the whole application will need to stop. This method returns as soon as the twisted thread has been started, which means the application must wait for the SIPApplicationDidStart notification in order to know that the application started.

```
stop(self)
```

Stop all the components started by the SIPApplication. This method returns immediately, but a SIPApplicationDidEnd notification is sent when all the components have been stopped.

attributes

`running`

True if the SIPApplication is running (it has been started and it has not been told to stop), False otherwise.

`storage`

Holds an object which implements the ISIPSimpleStorage interface which will be used to provide a storage facility to other middleware components.

`local_nat_type`

String containing the detected local NAT type.

`alert_audio_mixer`

The AudioMixer object created on the alert audio device as defined by the configuration (by SIPSimpleSettings.audio.alert_device).

`alert_audio_bridge`

An AudioBridge where IAudioPort objects can be added to playback sound to the alert device.

`alert_audio_device`

An AudioDevice which corresponds to the alert device as defined by the configuration. This will always be part of the alert_audio_bridge.

`voice_audio_mixer`

The AudioMixer object created on the voice audio device as defined by the configuration (by SIPSimpleSettings.audio.input_device and SIPSimpleSettings.audio.output_device).

`voice_audio_bridge`

An AudioBridge where IAudioPort objects can be added to playback sound to the output device or record sound from the input device.

`voice_audio_device`

An AudioDevice which corresponds to the voice device as defined by the configuration. This will always be part of the voice_audio_bridge.

notifications

`SIPApplicationWillStart`

This notification is sent just after the configuration has been loaded and the twisted thread started, but before any other components have been initialized.

timestamp:

A datetime.datetime object indicating when the notification was sent.

`SIPApplicationDidStart`

This notification is sent when all the components have been initialized. Note: it doesn't mean that all components have succeeded, for example, the account might not have registered by this time, but the registration process will have started.

timestamp:

A datetime.datetime object indicating when the notification was sent.

`SIPApplicationWillEnd`

This notification is sent as soon as the stop() method has been called.

timestamp:

A datetime.datetime object indicating when the notification was sent.

`SIPApplicationDidEnd`

This notification is sent when all the components have been stopped. All components have been given reasonable time to shutdown gracefully, such as the account unregistering. However, because of factors outside the control of the middleware, such as network problems, some components might not have actually shutdown gracefully; this is needed because otherwise the SIPApplication could hang indefinitely (for example because the system is no longer connected to a network and it cannot be determined when it will be again).

timestamp:

A datetime.datetime object indicating when the notification was sent.

`SIPApplicationFailedToStartTLS`

This notification is sent when a problem arises with initializing the TLS transport. In this case, the Engine will be started without TLS support and this notification contains the error which identifies the cause for not being able to start the TLS transport.

timestamp:

A datetime.datetime object indicating when the notification was sent.

error:

The exception raised by the Engine which identifies the cause for not being able to start the TLS transport.

Storage API

Different middleware components may need to store data, i.e. configuration files or XCAP documents. The Storage API defines a collection of backends which other components will use to store their data.

API Definition

The Storage API currently requires the following attributes to be defined as per the ISIPSimpleStorage interface:

```
configuration_backend
```

The backend used for storing the configuration.

```
xcap_storage_factory
```

Factory used to create XCAP storage backends for each account.

Provided implementations

Two storage implementations are provided: FileStorage and MemoryStorage both located in the sipsimple.storage module.

SIP Sessions

SIP sessions are supported by the sipsimple.session.Session class and independent stream classes, which need to implement the sipsimple.streams.IMediaStream interface.

The Session class takes care of the signalling, while the streams offer the actual media support which is negotiated by the Session.

The streams which are implemented in the SIP SIMPLE middleware are provided in modules within the sipsimple.streams package, but they are accessible for import directly from sipsimple.streams. Currently, the middleware implements two types of streams, one for RTP data, with a concrete implementation in the AudioStream class, and one for MSRP sessions, with concrete implementations in the ChatStream, FileTransferStream and DesktopSharingStream classes. However, the application can provide its own stream implementation, provided they respect the IMediaStream interface.

The sipsimple.streams module also provides a mechanism for automatically registering media streams in order for them to be used for incoming sessions. This is explained in more detail in MediaStreamRegistry.

SessionManager

```
Implemented in sipsimple/session.py
```

The sipsimple.session.SessionManager class is a singleton, which acts as the central aggregation point for sessions within the middleware. Although it is mainly used internally, the application can use it to query information about all active sessions.

The SessionManager is implemented as a singleton, meaning that only one instance of this class exists within the middleware. The SessionManager is started by the SIPApplication and takes care of handling incoming sessions and closing all sessions when SIPApplication is stopped.

attributes

```
sessions
```

A property providing a copy of the list of all active Sesssion objects within the application, meaning any Session object that exists globally within the application and is not in the NULL or TERMINATED state.

methods

```
__init__(self)
```

Instantiate a new SessionManager object.

```
start(self)
```

Start the SessionManager in order to be able to handle incoming sessions. This method is called automatically when SIPApplication is started. The application should not call this method directly.

```
stop(self)
```

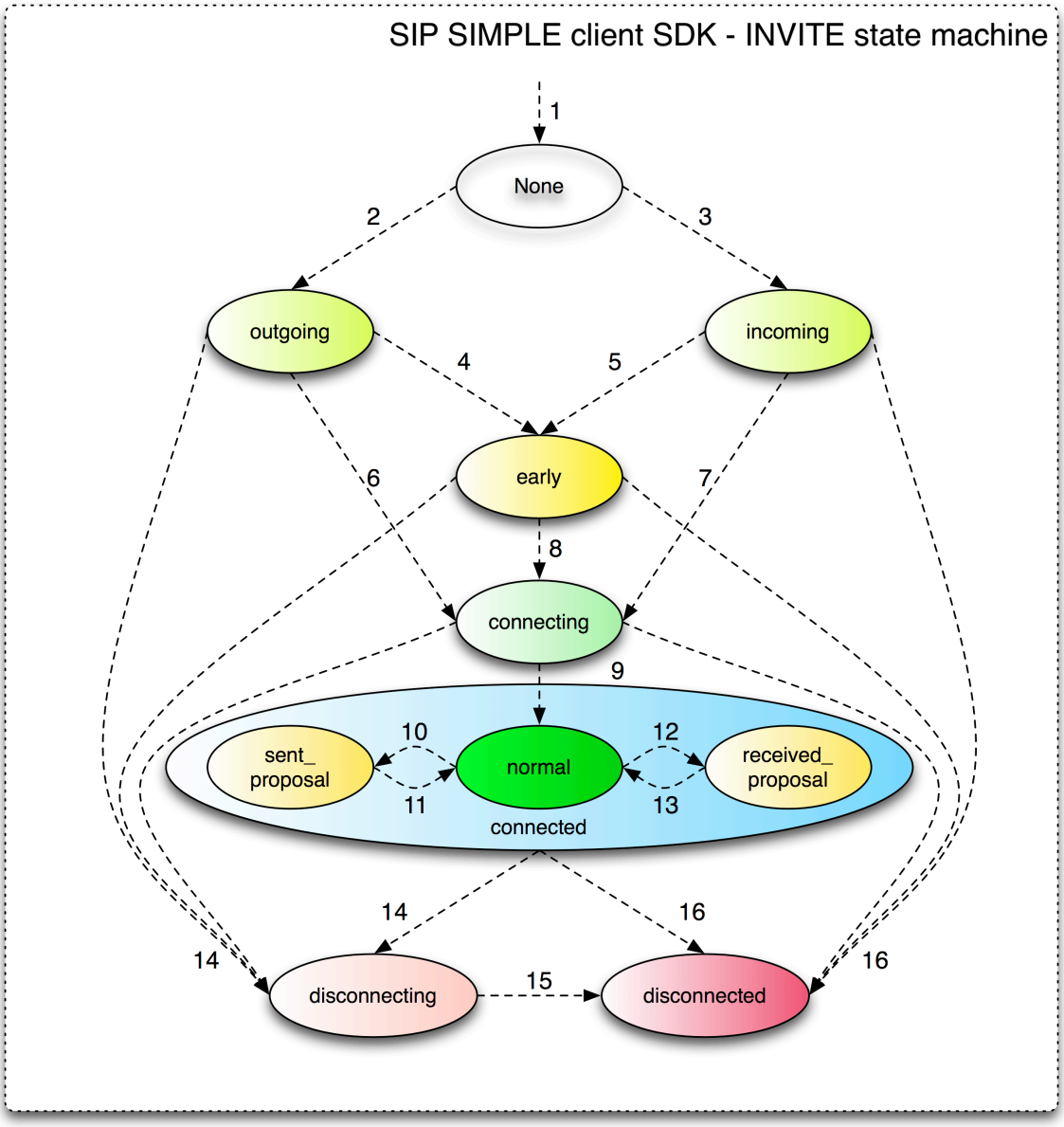
End all connected sessions. This method is called automatically when SIPApplication is stopped. The application should not call this method directly.

Session

```
Implemented in sipsimple/session.py
```

A sipsimple.session.Session object represents a complete SIP session between the local and a remote endpoints. Both incoming and outgoing sessions are represented by this class.

A Session instance is a stateful object, meaning that it has a state attribute and that the lifetime of the session traverses different states, from session creation to termination. State changes are triggered by methods called on the object by the application or by received network events. These states and their transitions are represented in the following diagram:



Although these states are crucial to the correct operation of the Session object, an application using this object does not need to keep track of these states, as a set of notifications is also emitted, which provide all the necessary information to the application.

The Session is completely independent of the streams it contains, which need to be implementations of the sipsimple.streams.IMediaStream interface. This interface provides the API by which the Session communicates with the streams. This API should not be used by the application, unless it also provides stream implementations or a SIP INVITE session implementation.

methods

```
__init__(self, account)
```

Creates a new Session object in the None state.

account:

The local account to be associated with this Session.

```
connect(self, to_header, routes, streams, is_focus=False,
subject=None)
```

Will set up the Session as outbound and propose the new session to the specified remote party and move the state machine to the outgoing state. Before contacting the remote party, a SIPSessionNewOutgoing notification will be emitted. If there is a failure or the remote party rejected the offer, a SIPSessionDidFail notification will be sent. Any time a ringing indication is received from the remote party, a SIPSessionGotRingIndication notification is sent. If the remote party accepted the session, a SIPSessionWillStart notification will be sent, followed by a SIPSessionDidStart notification when the session is actually established. This method may only be called while in the None state.

to_header:

A sipsimple.core.ToHeader object representing the remote identity to initiate the session to.

routes:

An iterable of sipsimple.util.Route objects, specifying the IP, port and transport to the outbound proxy. These routes will be tried in order, until one of them succeeds.

streams:

A list of stream objects which will be offered to the remote endpoint.

is_focus:

Boolean flag indicating if the isfocus parameter should be added to the Contact header according to RFC 4579.

subject:

Session subject. If not None a Subject header will be added with the specified value.

```
send_ring_indication(self)
```

Sends a 180 provisional response in the case of an incoming session.

```
accept(self, streams)
```

Calling this methods will accept an incoming session and move the state machine to the accepting state. When there is a new incoming session, a SIPSessionNewIncoming notification is sent, after which the application can call this method on the sender of the notification. After this method is called, SIPSessionWillStart followed by SIPSessionDidStart will be emitted, or SIPSessionDidFail on an error. This method may only be called while in the incoming state.

streams:

A list of streams which needs to be a subset of the proposed streams which indicates which streams are to be accepted. All the other proposed streams will be rejected.

```
reject(self, code=603, reason=None)
```

Reject an incoming session and move it to the terminaing state, which eventually leads to the terminated state. Calling this method will cause the Session object to emit a SIPSessionDidFail notification once the session has been rejected. This method may only be called while in the incoming state.

code:

An integer which represents the SIP status code in the response which is to be sent. Usually, this is either 486 (Busy) or 603 (Decline/Busy Everywhere).

reason:

The string which is to be sent as the SIP status reason in the response, or None if PJSIP's default reason for the specified code is to be sent.

```
accept_proposal(self, streams)
```

When the remote party proposes to add some new streams, signaled by the SIPSessionGotProposal notification, the application can use this method to accept the stream(s) being proposed. After calling this method a SIPSessionGotAcceptProposal notification is sent, unless an error occurs while setting up the new stream, in which case a SIPSessionHadProposalFailure notification is sent and a rejection is sent to the remote party. As with any action which causes the streams in the session to change, a SIPSessionDidRenegotiateStreams notification is also sent. This method may only be called while in the received_proposal state.

streams:

A list of streams which needs to be a subset of the proposed streams which indicates which streams are to be accepted. All the other proposed streams will be rejected.

```
reject_proposal(self, code=488, reason=None)
```

When the remote party proposes new streams that the application does not want to accept, this method can be used to reject the proposal, after which a SIPSessionGotRejectProposal or SIPSessionHadProposalFailure notification is sent. This method may only be called while in the received_proposal state.

code:

An integer which represents the SIP status code in the response which is to be sent. Usually, this is 488 (Not Acceptable Here).

reason:

The string which is to be sent as the SIP status reason in the response, or None if PJSIP's default reason for the specified code is to be sent.

```
add_stream(self, stream)
```

Proposes a new stream to the remote party. Calling this method will cause a SIPSessionGotProposal notification to be emitted. After this, the state machine will move into the sending_proposal state until either a SIPSessionGotAcceptProposal, SIPSessionGotRejectProposal or SIPSessionHadProposalFailure notification is sent, informing the application if the remote party accepted the proposal. As with any action which causes the streams in the session to change, a SIPSessionDidRenegotiateStreams notification is also sent. This method may only be called while in the connected state.

```
remove_stream(self, stream)
```

Stop the stream and remove it from the session, informing the remote party of this. Although technically this is also done via an SDP negotiation which may fail, the stream will always get remove (if the remote party refuses the re-INVITE, the result will be that the remote party will have a different view of the active streams than the local party). This method may only be called while in the connected state.

```
cancel_proposal(self)
```

This method cancels a proposal of adding a stream to the session by sending a CANCEL request. A SIPSessionGotRejectProposal notification will be sent with code 487.

```
hold(self)
```

Put the streams of the session which support the notion of hold on hold. This will cause a SIPSessionDidChangeHoldState notification to be sent. This method may be called in any state and will send the re-INVITE as soon as it is possible.

```
unhold(self)
```

Take the streams of the session which support the notion of hold out of hold. This will cause a SIPSessionDidChangeHoldState notification to be sent. This method may be called in any state and will send teh re-INVITE as soon as it is possible.

```
end(self)
```

This method may be called any time after the Session has started in order to terminate the session by sending a BYE request. Right before termination a SIPSessionWillEnd notification is sent, after termination SIPSessionDidEnd is sent.

attributes

```
state
```

The state the object is currently in, being one of the states from the diagram above.

```
account
```

The sipsimple.account.Account or sipsimple.account.BonjourAccount object that the Session is associated with. On an outbound session, this is the account the application specified on object instantiation.

```
direction
```

A string indicating the direction of the initial negotiation of the session. This can be either None, "incoming" or "outgoing".

```
transport
```

A string representing the transport this Session is using: "udp", "tcp" or "tls".

```
start_time
```

The time the session started as a datetime.datetime object, or None if the session was not yet started.

```
stop_time
```

The time the session stopped as a datetime.datetime object, or None if the session has not yet terminated.

```
on_hold
```

Boolean indicating whether the session was put on hold, either by the local or the remote party.

```
remote_user_agent
```

A string indicating the remote user agent, if it provided one. Initially this will be None, it will be set as soon as this information is received from the remote party (which may be never).

```
local_identity
```

The sipsimple.core.FrozenFromHeader or sipsimple.core.FrozenToHeader identifying the local party, if the session is active, None otherwise.

```
remote_identity
```

The sipsimple.core.FrozenFromHeader or sipsimple.core.FrozenToHeader identifying the remote party, if the session is active, None otherwise.

```
streams
```

A list of the currently active streams in the Session.

```
proposed_streams
```

A list of the currently proposed streams in the Session, or None if there is no proposal in progress.

```
conference
```

A ConferenceHandler object instance (or Null). It can be later used to add/remove participants from a remote conference.

```
subject
```

The session subject as a unicode object.

notifications

```
SIPSessionNewIncoming
```

Will be sent when a new incoming Session is received. The application should listen for this notification to get informed of incoming sessions.

timestamp:

A datetime.datetime object indicating when the notification was sent.

streams:

A list of streams that were proposed by the remote party.

```
SIPSessionNewOutgoing
```

Will be sent when the application requests a new outgoing Session.

timestamp:

A datetime.datetime object indicating when the notification was sent.

streams:

A list of streams that were proposed to the remote party.

```
SIPSessionGotRingIndication
```

Will be sent when an outgoing Session receives an indication that a remote device is ringing.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPSessionGotProvisionalResponse
```

Will be sent whenever the Session receives a provisional response as a result of sending a (re-)INVITE.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The SIP status code received.

reason:

The SIP status reason received.

```
SIPSessionWillStart
```

Will be sent just before a Session completes negotiation. In terms of SIP, this is sent after the final response to the INVITE, but before the ACK.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPSessionDidStart
```

Will be sent when a Session completes negotiation and all the streams have started. In terms of SIP this is sent after the ACK was sent or received.

timestamp:

A datetime.datetime object indicating when the notification was sent.

streams:

The list of streams which now form the active streams of the Session.

```
SIPSessionDidFail
```

This notification is sent whenever the session fails before it starts. The failure reason is included in the data attributes. This notification is never followed by SIPSessionDidEnd.

timestamp:

A datetime.datetime object indicating when the notification was sent.

originator:

A string indicating the originator of the Session. This will either be "local" or "remote".

code:

The SIP error code of the failure.

reason:

A SIP status reason.

failure_reason:

A string which represents the reason for the failure, such as "user_request", "missing ACK", "SIP core error...".

```
SIPSessionWillEnd
```

Will be sent just before terminating a Session.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPSessionDidEnd
```

Will be sent always when a Session ends as a result of remote or local session termination.

timestamp:

A datetime.datetime object indicating when the notification was sent.

originator:

A string indicating who originated the termination. This will either be "local" or "remote".

end_reason:

A string representing the termination reason, such as "user_request", "SIP core error...".

```
SIPSessionDidChangeHoldState
```

Will be sent when the session got put on hold or removed from hold, either by the local or the remote party.

timestamp:

A datetime.datetime object indicating when the notification was sent.

originator:

A string indicating who originated the hold request, and consequently in which direction the session got put on hold.

on_hold:

True if there is at least one stream which is on hold and False otherwise.

partial:

True if there is at least one stream which is on hold and one stream which supports hold but is not on hold and False otherwise.

```
SIPSessionGotProposal
```

Will be sent when either the local or the remote party proposes to add streams to the session.

timestamp:

A datetime.datetime object indicating when the notification was sent.

originator:

The party that initiated the stream proposal, can be either "local" or "remote".

streams:

A list of streams that were proposed.

```
SIPSessionGotRejectProposal
```

Will be sent when either the local or the remote party rejects a proposal to have streams added to the session.

timestamp:

A datetime.datetime object indicating when the notification was sent.

originator:

The party that initiated the stream proposal, can be either "local" or "remote".

code:

The code with which the proposal was rejected.

reason:

The reason for rejecting the stream proposal.

streams:

The list of streams which were rejected.

```
SIPSessionGotAcceptProposal
```

Will be sent when either the local or the remote party accepts a proposal to have stream(added to the session.

timestamp:

A datetime.datetime object indicating when the notification was sent.

originator:

The party that initiated the stream proposal, can be either "local" or "remote".

streams:

The list of streams which were accepted.

proposed_streams:

The list of streams which were originally proposed.

```
SIPSessionHadProposalFailure
```

Will be sent when a re-INVITE fails because of an internal reason (such as a stream not being able to start).

timestamp:

A datetime.datetime object indicating when the notification was sent.

failure_reason:

The error which caused the proposal to fail.

streams:

THe streams which were part of this proposal.

```
SIPSessionDidRenegotiateStreams
```

Will be sent when a media stream is either activated or deactivated. An application should listen to this notification in order to know when a media stream can be used.

timestamp:

A datetime.datetime object indicating when the notification was sent.

action:

A string which is either "add" or "remove" which specifies what happened to the streams the notificaton referes to

streams:

A list with the streams which were added or removed.

```
SIPSessionDidProcessTransaction
```

Will be sent whenever a SIP transaction is complete in order to provide low-level details of the progress of the INVITE dialog.

timestamp:

A datetime.datetime object indicating when the notification was sent.

originator:

The initiator of the transaction, "local" or "remote".

method:

The method of the request.

code:

The SIP status code of the response.

reason:

The SIP status reason of the response.

ack_received:

This attribute is only present for INVITE transactions and has one of the values True, False or "unknown". The last value may occur then PJSIP does not let us know whether the ACK was received or not.

IMediaStream

```
Implemented in sipsimple/streams/__init__.py
```

This interface describes the API which the Session uses to communicate with the streams. All streams used by the Session must respect this interface.

methods

```
__init__(self, account)
```

Initializes the generic stream instance.

```
new_from_sdp(cls, account, remote_sdp, stream_index)
```

A classmethod which returns an instance of this stream implementation if the sdp is accepted by the stream or None otherwise.

account:

The sipsimple.account.Account or sipsimple.account.BonjourAccount object the session which this stream would be part of is associated with.

remote_sdp:

The FrozenSDPSession which was received by the remote offer.

stream_index:

An integer representing the index within the list of media streams within the whole SDP which this stream would be instantiated for.

```
get_local_media(self, for_offer)
```

Return an SDPMediaStream which represents an offer for using this stream if for_offer is True and a response to an SDP proposal otherwise.

for_offer:

True if the SDPMediaStream will be used for an SDP proposal and False if for a response.

```
initialize(self, session, direction)
```

Initializes the stream. This method will get called as soon as the stream is known to be at least offered as part of the Session. If initialization goes fine, the stream must send a MediaStreamDidInitialize notification or a MediaStreamDidFail notification otherwise.

session:

The Session object this stream will be part of.

direction:

"incoming" if the stream was created because of a received proposal and "outgoing" if a proposal was sent. Note that this need not be the same as the initial direction of the Session since streams can be proposed in either way using re-INVITEs.

```
start(self, local_sdp, remote_sdp, stream_index)
```

Starts the stream. This method will be called as soon is known to be used in the Session (eg. only called for an incoming proposal if the local party accepts the proposed stream). If starting succeeds, the stream must send a MediaStreamDidStart notification or a MediaStreamDidFail notification otherwise.

local_sdp:

The FrozenSDPSession which is used by the local endpoint.

remote_sdp:

The FrozenSDPSession which is used by the remote endpoint.

stream_index:

An integer representing the index within the list of media streams within the whole SDP which this stream is represented by.

```
validate_update(self, remote_sdp, stream_index)
```

This method will be called when a re-INVITE is received which changes the parameters of the stream within the SDP. The stream must return True if the changes are acceptable or False otherwise. If any changed streams return False for a re-INVITE, the re-INVITE will be refused with a negative response. This means that streams must not changed any internal data when this method is called as the update is not guaranteed to be applied even if the stream returns True.

remote_sdp:

The FrozenSDPSession which is used by the remote endpoint.

stream_index:

An integer representing the index within the list of media streams within the whole SDP which this stream is represented by.

```
update(self, local_sdp, remote_sdp, stream_index)
```

This method is called when the an SDP negotiation initiated by either the local party or the remote party succeeds. The stream must update its internal state according to the new SDP in use.

local_sdp:

The FrozenSDPSession which is used by the local endpoint.

remote_sdp:

The FrozenSDPSession which is used by the remote endpoint.

stream_index:

An integer representing the index within the list of media streams within the whole SDP which this stream is represented by.

```
hold(self)
```

Puts the stream on hold if supported by the stream. Typically used by audio and video streams. The stream must immediately stop sending/receiving data and calls to get_local_media() following calls to this method must return an SDP which reflects the new hold state.

```
unhold(self)
```

Takes the stream off hold. Typically used by audio and video streams. Calls to get_local_media() following calls to this method must return an SDP which reflects the new hold state.

```
deactivate(self)
```

This method is called on a stream just before the stream will be removed from the Session (either as a result of a re-INVITE or a BYE). This method is needed because it avoids a race condition with streams using stateful protocols such as TCP: the stream connection might be terminated before the SIP signalling announces this due to network routing inconsistencies and the other endpoint would not be able to distinguish between this case and an error which caused the stream transport to fail. The stream must not take any action, but must consider that the transport being closed by the other endpoint after this method was called as a normal situation rather than an error condition.

```
end(self)
```

Ends the stream. This must close the underlying transport connection. The stream must send a MediaStreamWillEnd just after this method is called and a MediaStreamDidEnd as soon as the operation is complete. This method is always be called by the Session on the stream if at least the initialize() method has been called. This means that once a stream sends the MediaStreamDidFail notification, the Session will still call this method.

attributes

```
type (class attribute)
```

A string identifying the stream type (eg: "audio", "video").

```
priority (class attribute)
```

An integer value indicating the stream priority relative to the other streams types (higher numbers have higher priority).

```
hold_supported
```

True if the stream supports hold

```
on_hold_by_local
```

True if the stream is on hold by the local party

```
on_hold_by_remote
```

True if the stream is on hold by the remote

```
on_hold
```

True if either on_hold_by_local or on_hold_by_remote is true

notifications

These notifications must be generated by all streams in order for the Session to know the state of the stream.

```
MediaStreamDidInitialize
```

Sent when the stream has been successfully initialized.

```
MediaStreamDidStart
```

Sent when the stream has been successfully started.

```
MediaStreamDidFail
```

Sent when the stream has failed either as a result of calling one of its methods, or during the normal operation of the stream (such as the transport connection being closed).

```
MediaStreamWillEnd
```

Sent immediately after the end() method is called.

```
MediaStreamDidEnd
```

Sent when the end() method finished closing the stream.

MediaStreamRegistry

The MediaStream registry is a collection of classes which implement IMediaStream. This collection is used by the Session to select a stream class for instantiation in the case of an incomming session. The streams are included in the collection in the descending order of their priority. Thus, streams with a higher priority will be tried first by the Session. This object is a Singleton so references to the same object can be obtained by a simple instantiation.

There are several pre-built streams based on the IMediaStream API:

sipsimple.streams.rtp.AudioStream

Audio stream based on RTP

sipsimple.streams.msrp.ChatStream

Chat stream based on MSRP

sipsimple.streams.msrp.FileTransferStream

File Transfer stream based on MSRP

sipsimple.streams.msrp.DesktopSharingStream

Desktop Sharing stream based on VNC over MSRP

Other streams which are created by the application must be registered in this registry. For a simple way of doing this, see MediaStreamRegistrar.

methods

```
__init__(self)
```

Instantiate the MediaStreamRegistry. This will be called just once when first (and only) instance is created.

```
__iter__(self)
```

This method allows the registry to be iterated through and will return classes which were registered to it.

```
add(self, cls)
```

Add cls to the registry of streams. The class must implement the IMediaStream interface.

MediaStreamRegistrar

This is a convenience metaclass which automatically registers a defined class with the MediaStreamRegistry. In order to use this class, one simply needs to use it as the metaclass of the new stream.

```
from zope.interface import implements
from sipsimple.streams import IMediaStream, MediaStreamRegistrar

class MyStream(object):
__metaclass__ = MediaStreamRegistrar

implements(IMediaStream)
[...]
```

AudioStream

```
Implemented in sipsimple/streams/rtp.py
```

The AudioStream is an implementation of IMediaStream which supports audio data using the AudioTransport and RTPTransport of the SIP core. As such, it provides all features of these objects, including ICE negotiation. An example SDP created using the AudioStream is provided below:

```
Content-Type: application/sdp
Content-Length: 1093

v=0
o=- 3467525278 3467525278 IN IP4 192.168.1.6
s=blink-0.10.7-beta
c=IN IP4 80.101.96.20
t=0 0
m=audio 55328 RTP/AVP 104 103 102 3 9 0 8 101
a=rtcp:55329 IN IP4 80.101.96.20
a=rtpmap:104 speex/32000
a=rtpmap:103 speex/16000
a=rtpmap:102 speex/8000
a=rtpmap:3 GSM/8000
a=rtpmap:9 G722/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=crypto:1 AES_CM_128_HMAC_SHA1_80
inline:esI6DbLY1+Aceu0JNswN9Z10DcFx5cZwqJcu91jb
a=crypto:2 AES_CM_128_HMAC_SHA1_32
inline:SHuEMm1BYJqOF4udKl73EaCwnsI57pO86bYKsg70
a=ice-ufrag:2701ed80
a=ice-pwd:6f8f8281
a=candidate:S 1 UDP 31 80.101.96.20 55328 typ srflx raddr 192.168.1.6
rport 55328
a=candidate:H 1 UDP 23 192.168.1.6 55328 typ host
a=candidate:H 1 UDP 23 10.211.55.2 55328 typ host
a=candidate:H 1 UDP 23 10.37.129.2 55328 typ host
a=candidate:S 2 UDP 30 80.101.96.20 55329 typ srflx raddr 192.168.1.6
rport 55329
a=candidate:H 2 UDP 22 192.168.1.6 55329 typ host
a=candidate:H 2 UDP 22 10.211.55.2 55329 typ host
a=candidate:H 2 UDP 22 10.37.129.2 55329 typ host
a=sendrecv
```

As an implementation of IAudioPort, an AudioStream can be added to an AudioBridge to send or to read audio data to/from other audio objects. It is connected to the voice AudioMixer (SIPApplication.voice_audio_mixer) so it can only be added to bridges using the same AudioMixer. It also contains an AudioBridge on the bridge attribute which always contains an AudioDevice corresponding to the input and output devices; when the stream is active (started and not on hold), the bridge also

contains the stream itself and when recording is active, the stream contains a WaveRecorder which records audio data.

methods

```
start_recording(self, filename=None)
```

If an audio stream is present within this session, calling this method will record the audio to a .wav file. Note that when the session is on hold, nothing will be recorded to the file. Right before starting the recording a SIPSessionWillStartRecordingAudio notification will be emitted, followed by a SIPSessionDidStartRecordingAudio. This method may only be called while the stream is started.

filename:

> The name of the .wav file to record to. If this is set to None, a default file name including the session participants and the timestamp will be generated using the directory defined in the configuration.

```
stop_recording(self)
```

This will stop a previously started recording. Before stopping, a SIPSessionWillStopRecordingAudio notification will be sent, followed by a SIPSessionDidStopRecordingAudio.

```
send_dtmf(self, digit)
```

If the audio stream is started, sends a DTMF digit to the remote party.

digit:

> This should a string of length 1, containing a valid DTMF digit value (0-9, A-D, * or #).

attributes

```
sample_rate
```

If the audio stream was started, this attribute contains the sample rate of the audio negotiated.

```
codec
```

If the audio stream was started, this attribute contains the name of the audio codec that was negotiated.

```
srtp_active
```

If the audio stream was started, this boolean attribute indicates if SRTP is currently being used on the stream.

`ice_active`

True if the ICE candidates negotiated are being used, False otherwise.

`local_rtp_address`

If an audio stream is present within the session, this attribute contains the local IP address used for the audio stream.

`local_rtp_port`

If an audio stream is present within the session, this attribute contains the local UDP port used for the audio stream.

`remote_rtp_address_sdp`

If the audio stream was started, this attribute contains the IP address that the remote party gave to send audio to.

`remote_rtp_port_sdp`

If the audio stream was started, this attribute contains the UDP port that the remote party gave to send audio to.

`remote_rtp_address_received`

If the audio stream was started, this attribute contains the remote IP address from which the audio stream is being received.

`remote_rtp_port_received`

If the audio stream was started, this attribute contains the remote UDP port from which the audio stream is being received.

`local_rtp_candidate_type`

The local ICE candidate type which was selected by the ICE negotiation if it succeeded and None otherwise.

`remote_rtp_candidate_type`

The remote ICE candidate type which was selected by the ICE negotiation if it succeeded and None otherwise.

`recording_filename`

If the audio stream is currently being recorded to disk, this property contains the name of the .wav file being recorded to.

notifications

```
AudioStreamDidChangeHoldState
```

Will be sent when the hold state is changed as a result of either a SIP message received on the network or the application calling the hold()/unhold() methods on the Session this stream is part of.

timestamp:

A datetime.datetime object indicating when the notification was sent.

originator:

A string representing the party which requested the hold change, "local" or "remote"

on_hold:

A boolean indicating the new hold state from the point of view of the originator.

```
AudioStreamWillStartRecordingAudio
```

Will be sent when the application requested that the audio stream be recorded to a .wav file, just before recording starts.

timestamp:

A datetime.datetime object indicating when the notification was sent.

filename:

The full path to the .wav file being recorded to.

```
AudioStreamDidStartRecordingAudio
```

Will be sent when the application requested that the audio stream be recorded to a .wav file, just after recording started.

timestamp:

A datetime.datetime object indicating when the notification was sent.

filename:

The full path to the .wav file being recorded to.

```
AudioStreamWillStopRecordingAudio
```

Will be sent when the application requested ending the recording to a .wav file, just before recording stops.

timestamp:

A datetime.datetime object indicating when the notification was sent.

filename:

The full path to the .wav file being recorded to.

```
AudioStreamDidStopRecordingAudio
```

Will be sent when the application requested ending the recording to a .wav file, just after recording stoped.

timestamp:

A datetime.datetime object indicating when the notification was sent.

filename:

The full path to the .wav file being recorded to.

```
AudioStreamDidChangeRTPParameters
```

This notification is sent when the RTP parameters are changed, such as codec, sample rate, RTP port etc.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
AudioStreamGotDTMF
```

Will be send if there is a DMTF digit received from the remote party on the audio stream.

timestamp:

A datetime.datetime object indicating when the notification was sent.

digit:

The DTMF digit that was received, in the form of a string of length 1.

```
AudioStreamICENegotiationStateDidChange
```

This notification is proxied from the RTPTransport and as such has the same data as the RTPTransportICENegotiationStateDidChange.

```
AudioStreamICENegotiationDidSucceed
```

This notification is proxied from the RTPTransport and as such has the same data as the RTPTransportICENegotiationDidSucceed.

```
AudioStreamICENegotiationDidFail
```

This notification is proxied from the RTPTransport and as such has the same data as the RTPTransportICENegotiationDidFail.

```
AudioStreamDidTimeout
```

This notification is proxied from the RTPTransport. It's sent when the RTP transport did not receive any data after the specified amount of time (rtp.timeout setting in the Account).

MSRPStreamBase

```
Implemented in sipsimple/streams/msrp.py
```

The MSRPStreamBase is used as a base class for streams using the MSRP protocol. Within the SIP SIMPLE middleware, this hold for the ChatStream, FileTransferStream and DesktopSharingStream classes, however the application can also make use of this class to implement some other streams based on the MSRP protocol as a transport.

methods

Of the methods defined by the IMediaStream interface, only the new_from_sdp method is not implemented in this base class and needs to be provided by the subclasses. Also, the subclasses can defined methods of the form _handle_XXX, where XXX is a MSRP method name in order to handle incoming MSRP requests. Also, since this class registers as an observer for itself, it will receive the notifications it sends so subclasses can define methods having the signature _NH_<notification name>(self, notification) as used throughout the middleware in order to do various things at the different points within the life-cycle of the stream.

attributes

The attributes defined in the IMediaStream interface which are not provided by this class are:

1. type
2. priority

In addition, the following attributes need to be defined in the subclass in order for the MSRPStreamBase class to take the correct decisions

```
media_type
```

The media type as included in the SDP (eg. "message", "application").

```
accept_types
```

A list of the MIME types which should be accepted by the stream (this is also sent within the SDP).

```
accept_wrapped_types
```

A list of the MIME types which should be accepted by the stream while wrapped in a message/cpim envelope.

```
use_msrp_session
```

A boolean indicating whether or not an MSRPSession should be used.

notifications

While not technically notifications of MSRPStreamBase, these notifications are sent from the middleware on behalf of the MSRPTransport used by a stream in the former case, and anonymously in the latter.

```
MSRPTransportTrace
```

This notification is sent when an MSRP message is received for logging purposes.

timestamp:

A datetime.datetime object indicating when the notification was sent.

direction:

The direction of the message, "incoming" or "outgoing".

data:

The MSRP message as a string.

```
MSRPLibraryLog
```

This notification is sent anonymously whenever the MSRP library needs to log any information.

timestamp:

A datetime.datetime object indicating when the notification was sent.

message:

The log message as a string.

level:

The log level at which the message was written. One of the levels DEBUG, INFO, WARNING, ERROR, CRITICAL from the application.log.level object which is part of the python-application library.

ChatStream

```
Implemented in sipsimple/streams/msrp.py
```

sipsimple.streams.msrp.ChatStream implements session-based Instant Messaging (IM) over MSRP. This class performs the following functions:

1. automatically wraps outgoing messages with Message/CPIM if that's necessary according to accept-types
2. unwraps incoming Message/CPIM messages; for each incoming message, the ChatStreamGotMessage notification is posted
3. composes iscomposing payloads and reacts to those received by sending the ChatStreamGotComposingIndication notification

An example of an SDP created using this class follows:

```
Content-Type: application/sdp
Content-Length: 283

v=0
o=- 3467525214 3467525214 IN IP4 192.168.1.6
s=blink-0.10.7-beta
c=IN IP4 192.168.1.6
t=0 0
m=message 2855 TCP/TLS/MSRP *
a=path:msrps://192.168.1.6:2855/ca7940f12ddef14c3c32;tcp
a=accept-types:message/cpim text/* application/im-iscomposing+xml
a=accept-wrapped-types:*
```

methods

```
__init__(self, account, direction='sendrecv')
```

Initializes the ChatStream instance.

```
send_message(self, content, content_type='text/plain',
recipients=None, courtesy_recipients=None, subject=None,
timestamp=None, required=None, additional_headers=None)
```

Sends an IM message. Prefer Message/CPIM wrapper if it is supported. If called before the connection was established, the messages will be queued until the stream starts. Returns the generated MSRP message ID.

content:

The content of the message.

content_type:

Content-Type of wrapped message if Message/CPIM is used (Content-Type of MSRP message is always Message/CPIM in that case); otherwise, Content-Type of MSRP message.

recipients:

The list of CPIMIdentity objects which will be used for the To header of the CPIM wrapper. Used to override the default which depends on the remote identity. May only differ from the default one if the remote party supports private messages. If it does not, a ChatStreamError will be raised.

courtesy_recipients:

The list of CPIMIdentity objects which will be used for the cc header of the CPIM wrapper. May only be specified if the remote party supports private messages and CPIM is supported. If it does not, a ChatStreamError will be raised.

subject:

A string or MultilingualText which specifies the subject and its translations to be added to the CPIM message. If CPIM is not supported, a ChatStreamError will be raised.

required:

A list of strings describing the required capabilities that the other endpoint must support in order to understand this CPIM message. If CPIM is not supported, a ChatStreamError will be raised.

additional_headers:

A list of MSRP header objects which will be added to this CPIM message. If CPIM is not supported, a ChatStreamError will be raised.

timestamp:

A datetime.datetime object representing the timestamp to put on the CPIM wrapper of the message. When set to None, a default one representing the current moment will be added.

These MSRP headers are used to enable end-to-end success reports and to disable hop-to-hop successful responses:

1. Failure-Report: partial
2. Success-Report: yes

```
send_composing_indication(self, state, refresh, last_active=None,
recipients=None)
```

Sends an is-composing message to the listed recipients.

state:

The state of the endpoint, "active" or "idle".

refresh:

How often the local endpoint will send is-composing indications to keep the state from being reverted to "idle".

last_active:

A datatime.datetime object representing the moment when the local endpoint was last active.

recipients:

The list of CPIMIdentity objects which will be used for the To header of the CPIM wrapper. Used to override the default which depends on the remote identity. May only differ from the default one if the remote party supports private messages. If it does not, a ChatStreamError will be raised.

notifications

```
ChatStreamGotMessage
```

Sent whenever a new incoming message is received,

timestamp:

A datetime.datetime object indicating when the notification was sent.

message:

A ChatMessage or CPIMMessage instance, depending on whether a CPIM message was received or not.

```
ChatStreamDidDeliverMessage
```

Sent when a successful report is received.

timestamp:

A datetime.datetime object indicating when the notification was sent.

message_id:

Text identifier of the message.

code:

The status code received. Will always be 200 for this notification.

reason:

The status reason received.

chunk:

A msrplib.protocol.MSRPData instance providing all the MSRP information about the report.

```
ChatStreamDidNotDeliverMessage
```

Sent when a failure report is received.

timestamp:

A datetime.datetime object indicating when the notification was sent.

message_id:

Text identifier of the message.

code:

The status code received.

reason:

The status reason received.

chunk:

A msrplib.protocol.MSRPData instance providing all the MSRP information about the report.

```
ChatStreamDidSendMessage
```

Sent when an outgoing message has been sent.

timestamp:

A datetime.datetime object indicating when the notification was sent.

message:

A msrplib.protocol.MSRPData instance providing all the MSRP information about the sent message.

```
ChatStreamGotComposingIndication
```

Sent when a is-composing payload is received.

timestamp:

A datetime.datetime object indicating when the notification was sent.

state:

The state of the endpoint, "active" or "idle".

refresh:

How often the remote endpoint will send is-composing indications to keep the state from being reverted to "idle". May be None.

last_active:

A datatime.datetime object representing the moment when the remote endpoint was last active. May be None.

content_type:

The MIME type of message being composed. May be None.

sender:

The ChatIdentity or CPIMIdentity instance which identifies the sender of the is-composing indication.

recipients:

The ChatIdentity or CPIMIdentity instances list which identifies the recipients of the is-composing indication.

FileSelector

The FileSelector is used to contain information about a file tranfer using the FileTransferStream documented below.

methods

```
__init__(self, name=None, type=None, size=None, hash=None,
fd=None)
```

Instantiate a new FileSelector. All the arguments are also available as attributes.

name:

The filename (should be just the base name).

type:

The type of the file.

size:

The size of the file in bytes.

hash:

The hash of the file in the following format: hash:sha-1:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX, where X is a hexadecimal digit. Currently, only SHA1 hashes are supported according to the RFC.

fd:

A file descriptor if the application has already opened the file.

```
parse(cls, string)
```

Parses a file selector from the SDP file-selector a attribute and returns a FileSelector instance.

```
for_file(cls, path, content_type, compute_hash=True)
```

Returns a FileSelector instance for the specified file. The file identified by the path must exist. Note that if compute_hash is True this method will block while the hash is computed, a potentially long operation for large files.

path:

The full path to the file.

content_type:

An optional MIME type which is to be included in the file-selector.

compute_hash:

Whether or not this method should compute the hash of the file.

```
compute_hash(self)
```

Compute the hash for this file selector. This method will block while the hash is computed, a potentially long operation for large files.

attributes

```
sdp_repr
```

The SDP representation of the file-selector according to the RFC. This should be the value of the file-selector SDP attribute.

FileTransferStream

```
Implemented in sipsimple/streams/msrp.py
```

The FileTransferStream supports file transfer over MSRP according to RFC5547. An example of SDP constructed using this stream follows:

```
Content-Type: application/sdp
Content-Length: 383

v=0
o=- 3467525166 3467525166 IN IP4 192.168.1.6
s=blink-0.10.7-beta
c=IN IP4 192.168.1.6
t=0 0
m=message 2855 TCP/TLS/MSRP *
a=path:msrps://192.168.1.6:2855/e593357dc9abe90754bd;tcp
a=sendonly
a=accept-types:*
a=accept-wrapped-types:*
a=file-selector:name:"reblink.pdf" type:com.adobe.pdf size:268759
hash:sha1:60:A1:BE:8D:71:DB:E3:8E:84:C9:2C:62:9E:F2:99:78:9D:68:79:F6
```

methods

```
__init__(self, account, file_selector=None)
```

Instantiate a new FileTransferStream. If this is constructed by the application for an outgoing file transfer, the file_selector argument must be present.

account:

The sipsimple.account.Account or sipsimple.account.BonjourAccount instance which will be associated with the stream.

file_selector:

The FileSelector instance which represents the file which is to be transferred.

notifications

```
FileTransferStreamDidDeliverChunk
```

This notification is sent for an outgoing file transfer when a success report is received about part of the file being transferred.

timestamp:

A datetime.datetime object indicating when the notification was sent.

message_id:

The MSRP message ID of the file transfer session.

chunk:

An msrplib.protocol.MSRPData instance represented the received REPORT.

code:

The status code received. Will always be 200 for this notification.

reason:

The status reason received.

transferred_bytes:

The number of bytes which have currently been successfully transferred.

file_size:

The size of the file being transferred.

```
FileTransferStreamDidNotDeliverChunk
```

timestamp:

A datetime.datetime object indicating when the notification was sent. This notification is sent for an outgoing file transfer when a failure report is received about part of the file being transferred.

message_id:

The MSRP message ID of the file transfer session.

chunk:

An msrplib.protocol.MSRPData instance represented the received REPORT.

code:

The status code received.

reason:

The status reason received.

```
FileTransferStreamDidFinish
```

This notification is sent when the incoming or outgoing file transfer is finished.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
FileTransferStreamGotChunk
```

This notificaiton is sent for an incoming file transfer when a chunk of file data is received.

timestamp:

A datetime.datetime object indicating when the notification was sent.

content:

The file part which was received, as a str.

content_type:

The MIME type of the file which is being transferred.

transferred_bytes:

The number of bytes which have currently been successfully transferred.

file_size:

The size of the file being transferred.

IDesktopSharingHandler

This interface is used to describe the interface between a IDesktopSharingHandler, which is responsible for consuming and producing RFB data, and the DesktopSharingStream which is responsible for transporting the RFB data over MSRP. The middleware provides four implementations of this interface:

1. InternalVNCViewerHandler
2. InternalVNCServerHandler
3. ExternalVNCViewerHandler
4. ExternalVNCServerHandler

methods

```
initialize(self, stream)
```

This method will be called by the DesktopSharingStream when the stream has been started and RFB data can be transported. The stream has two attributes which are relevant to the IDesktopSharingHandler: incoming_queue and outgoing_queue. These attributes are eventlet.coros.queue instances which are used to transport RFB data between the stream and the handler.

attributes

```
type
```

"active" or "passive" depending on whether the handler represents a VNC viewer or server respectively.

notifications

```
DesktopSharingHandlerDidFail
```

This notification must be sent by the handler when an error occurs to notify the stream that it should fail.

context:

A string describing when the handler failed, such as "reading", "sending" or "connecting".

failure:

A twisted.python.failure.Failure instance describing the exception which led to the failure.

reason:

A string describing the failure reason.

InternalVNCViewerHandler

This is a concrete implementation of the IDesktopSharingHandler interface which can be used for a VNC viewer implemented within the application.

methods

```
send(self, data)
```

Sends the specified data to the stream in order for it to be transported over MSRP to the remote endpoint.

data:

The RFB data to be transported over MSRP, in the form of a str.

notifications

```
DesktopSharingStreamGotData
```

This notification is sent when data is received over MSRP.

data:

The RFB data from the remote endpoint, in the form of a str.

InternalVNCServerHandler

This is a concrete implementation of the IDesktopSharingHandler interface which can be used for a VNC server implemented within the application.

methods

```
send(self, data)
```

Sends the specified data to the stream in order for it to be transported over MSRP to the remote endpoint.

data:

The RFB data to be transported over MSRP, in the form of a str.

notifications

```
DesktopSharingStreamGotData
```

This notification is sent when data is received over MSRP.

data:

The RFB data from the remote endpoint, in the form of a str.

ExternalVNCViewerHandler

This implementation of IDesktopSharingHandler can be used for an external VNC viewer which connects to a VNC server using TCP.

methods

```
__init__(self, address=("localhost", 0), connect_timeout=3)
```

This instantiates a new ExternalVNCViewerHandler which is listening on the provided address, ready for the external VNC viewer to connect to it via TCP. After this method returns, the attribute address can be used to find out exactly on what address and port the handler is listening on. The handler will only accept one conenction on this address.

address:

> A tuple containing an IP address/hostname and a port on which the handler should listen. Any data received on this socket will then be forwarded to the stream and any data received from the stream will be forwarded to this socket.

attributes

```
address
```

A tuple containing an IP address and a port on which the handler is listening.

ExternalVNCServerHandler

This implementation of IDesktopSharingHandler can be used for an external VNC server to which handler will connect using TCP.

methods

```
__init__(self, address, connect_timeout=3)
```

This instantiates a new ExternalVNCServerHandler which will connect to the provided address on which a VNC server must be listening before the stream using this handler starts.

address:

A tuple containing an IP address/hostname and a port on which the VNC server will be listening. Any data received on this socket will then be forwared to the stream and any data received form the stream will be forwarded to this socket.

connect_timeout:

How long to wait to connect to the VNC server before giving up.

DesktopSharingStream

```
Implemented in sipsimple/streams/msrp.py
```

This stream implements desktop sharing using MSRP as a transport protocol for RFB data.

There is no standard defining this usage but is fairly easy to implement in clients that already support MSRP. To traverse a NAT-ed router, a MSRP relay configured for the called party domain is needed. Below is an example of the Session Description Protocol used for establishing a Desktop sharing session:

```
m=application 2855 TCP/TLS/MSRP *
a=path:msrps://10.0.1.19:2855/b599b22d1b1d6a3324c8;tcp
a=accept-types:application/x-rfb
a=rfbsetup:active
```

methods

```
__init__(self, acount, handler)
```

Instantiate a new DesktopSharingStream.

account:

The sipsimple.account.Account or sipsimple.account.BonjourAccount instance this stream is associated with.

handler:

An object implementing the IDesktopSharingHandler interface which will act as the handler for RFB data.

attributes

```
handler
```

This is a writable property which can be used to get or set the object implementing IDesktopSharingHandler which acts as the handler for RFB data. For incoming DesktopSharingStreams, this must be set by the application before the stream starts.

```
incoming_queue
```

A eventlet.coros.queue instance on which incoming RFB data is written. The handler should wait for data on this queue.

```
outgoing_queue
```

A eventlet.coros.queue instance on which outgoing RFB data is written. The handler should write data on this queue.

ConferenceHandler

This class is internal to the Session and provied the user with the ability to invite participants to a conference hosted by the remote endpoint.

Adding and removing participants is performed using a REFER request as explained in RFC 4579, section 5.5.

In addition, the ConferenceHandler will subscribe to the conference event in order to get information about participants in the conference.

methods

```
add_participant(self, participant_uri)
```

Send a REFER request telling the server to invite the participant specified in participant_uri to join the ongoing conference.

```
remove_participant(self, participant_uri)
```

Send a REFER request telling the server to remove the participant specified in participant_uri from the ongoing conference.

notifications

All notifications are sent with the Session object as the sender.

```
SIPSessionGotConferenceInfo
```

This notification is sent when a NOTIFY is received with a valid conferene payload.

timestamp:

A datetime.datetime object indicating when the notification was sent.

conference_info:

The Conference payload object.

```
SIPConferenceDidAddParticipant
```

This notification is sent when a participant was successfully added to the conference.

timestamp:

A datetime.datetime object indicating when the notification was sent.

participant:

URI of the participant added to the conference.

```
SIPConferenceDidNotAddParticipant
```

This notification is sent when a participant could not be added to the conference.

timestamp:

A datetime.datetime object indicating when the notification was sent.

participant:

URI of the participant added to the conference.

code:

SIP response code for the failure.

reason:

Reason for the failure.

```
SIPConferenceDidRemoveParticipant
```

This notification is sent when a participant was successfully removed from the conference.

timestamp:

A datetime.datetime object indicating when the notification was sent.

participant:

URI of the participant removed from the conference.

```
SIPConferenceDidNotRemoveParticipant
```

This notification is sent when a participant could not be removed from the conference.

timestamp:

A datetime.datetime object indicating when the notification was sent.
participant:
URI of the participant removed from the conference.

code:

SIP response code for the failure.

reason:

Reason for the failure.

```
SIPConferenceGotAddParticipantProgress
```

This notification is sent when a NOTIFY is received indicating the status of the add participant operation.

timestamp:

A datetime.datetime object indicating when the notification was sent.

participant:

URI of the participant whose operation is in progress.

code:

SIP response code for progress.

reason:

Reason associated with the response code.

```
SIPConferenceGotRemoveParticipantProgress
```

This notification is sent when a NOTIFY is received indicating the status of the remove participant operation.

timestamp:

A datetime.datetime object indicating when the notification was sent.
participant:
URI of the participant whose operation is in progress.

code:

SIP response code for progress.

reason:

Reason associated with the response code.

Address Resolution

The SIP SIMPLE middleware offers the sipsimple.lookup module which contains an implementation for doing DNS lookups for SIP proxies, MSRP relays, STUN servers and XCAP servers. The interface offers both an asynchronous and synchronous interface. The asynchronous interface is based on notifications, while the synchronous one on green threads. In order to call the methods in a asynchronous manner, you just need to call the method and wait for the notification which is sent on behalf of the DNSLookup instance. The notifications sent by the DNSLookup object are DNSLookupDidSucceed and DNSLookupDidFail. In order to call the methods in a synchronous manner, you need to call the wait method on the object returned by the methods of DNSLookup. This wait method needs to be called from a green thread and will either return the result of the lookup or raise an exception.

The DNSLookup object uses DNSManager, an object that will use the system nameservers and it will fallback to Google's nameservers (8.8.8.8 and 8.8.4.4) in case of failure.

DNS Manager

This object provides DNSLookup with the nameserver list that will be used to perform DNS lookups. It will probe the system local nameservers and check if they are able to do proper lookups (by querying sip2sip.info domain). If the local nameservers are not able to do proper lookups Google nameservers will be used and another probing operation will be scheduled. Local nameservers are always preferred.

methods

```
__init__(self)
```

Instantiate the DNSManager object (it's a Singleton).

```
start(self)
```

Start the DNSManager. It will start the probing process to determine the suitable nameservers to use.

```
stop(self)
```

Stop the DNS resolution probing.

notifications

```
DNSResolverDidInitialize
```

This notification is sent when the nameservers to use for probing (and further DNS lookups) have been set for the first time.

timestamp:

A datetime.datetime object indicating when the notification was sent.

nameservers:

The list of nameservers that was set on the DNS Manager.

```
DNSNameserversDidChange
```

This notification is sent when the nameservers to use for probing (and further DNS lookups) have changed as a result of the probing process.

timestamp:

A datetime.datetime object indicating when the notification was sent.

nameservers:

The list of nameservers that was set on the DNS Manager.

DNS Lookup

This object implements DNS lookup support for SIP proxies according to RFC3263 and MSRP relay and STUN server lookup using SRV records. The object initially does NS record queries in order to determine the authoritative nameservers for the domain requested; these authoritative nameservers will then be used for NAPTR, SRV and A record queries. If this fails, the locally configured nameservers are used. The reason for doing this is that some home routers have broken NAPTR and/or SRV query support.

methods

```
__init__(self)
```

Instantiate a new DNSLookup object.

```
lookup_service(self, uri, service, timeout=3.0, lifetime=15.0)
```

Perform an SRV lookup followed by A lookups for MSRP relays or STUN servers depending on the service parameter. If SRV queries on the uri.host domain fail, an A lookup is performed on it and the default port for the service is returned. Only the uri.host attribute is used. The return value is a list of (host, port) tuples.

uri:

A (Frozen)SIPURI from which the host attribute is used for the query domain.

service:

The service to lookup servers for, "msrprelay" or "stun".

timeout:

How many seconds to wait for a response from a nameserver.

lifetime:

How many seconds to wait for a response from all nameservers in total.

```
lookup_sip_proxy(self, uri, supported_transports, timeout=3.0,
lifetime=15.0)
```

Performs a RFC3263 compliant DNS lookup for a SIP proxy using the URI which is considered to point to a host if either the host attribute is an IP address, or the port is present. Otherwise, it is considered a domain for which NAPTR, SRV and A lookups are performed. If NAPTR or SRV queries fail, they fallback to using SRV and A queries. If the transport parameter is present in the URI, this will be used as far as it is part of the supported transports. If the URI has a sips schema, then only the TLS transport will be used as far as it doesn't conflict with the supported transports or the transport parameter. The return value is a list of Route objects containing the IP address, port and transport to use for routing in the order of preference given by the supported transports argument.

uri:

A (Frozen)SIPURI from which the host, port, parameters and secure attributes are used.

supported_transports:

A sublist of ['udp', 'tcp', 'tls'] in the application's order of preference.

timeout:

How many seconds to wait for a response from a nameserver.

lifetime:

How many seconds to wait for a response from all nameservers in total.

```
lookup_xcap_server(self, uri, timeout=3.0, lifetime=15.0)
```

Perform a TXT DNS query on xcap.<uri.host> and return all values of the TXT record which are URIs with a scheme of http or https. Only the uri.host attribute is used. The return value is a list of strings representing HTTP URIs.

uri:

A (Frozen)SIPURI from which the host attribute is used for the query domain.

timeout:

How many seconds to wait for a response from a nameserver.

lifetime:

How many seconds to wait for a response from all nameservers in total.

notifications

```
DNSLookupDidSucceed
```

This notification is sent when one of the lookup methods succeeds in finding a result.

timestamp:

A datetime.datetime object indicating when the notification was sent.

result:

The result of the DNS lookup in the format described in each method.

```
DNSLookupDidFail
```

This notification is sent when one of the lookup methods fails in finding a result.

timestamp:

A datetime.datetime object indicating when the notification was sent.

error:

A str object describing the error which resulted in the DNS lookup failure.

```
DNSLookupTrace
```

This notification is sent several times during a lookup process for each individual DNS query.

timestamp:

A datetime.datetime object indicating when the notification was sent.

query_type:

The type of the query, "NAPTR", "SRV", "A", "NS" etc.

query_name:

The name which was queried.

answer:

The answer returned by dnspython, or None if an error occurred.

error:

The exception which caused the query to fail, or None if no error occurred.

context:

The name of the method which was called on the DNSLookup object.

service:

The service which was queried for, only available when context is "lookup_service".

uri:

The uri which was queried for.

nameservers:

The list of nameservers that was used to perform the lookup.

Route

This is a convinience object which contains sufficient information to identify a route to a SIP proxy. This object is returned by DNSLookup.lookup_sip_proxy and can be used with the Session or a (Frozen)RouteHeader can be easily constructed from it to pass to one of the objects in the SIP core handling SIP dialogs/transactions (Invitation, Subscription, Request, Registration, Message, Publication).

This object has three attributes which can be set in the constructor or after it was instantiated. They will only be documented as arguments to the constructor.

methods

```
__init__(self, address, port=None, transport='udp')
```

Creates the Route object with the specified parameters as attributes. Each of these attributes can be accessed on the object once instanced.

address:

The IPv4 address that the request in question should be sent to as a string.

port:

The port to send the requests to, represented as an int, or None if the default port is to be used.

transport:

The transport to use, this can be a string of either "udp", "tcp" or "tls" (case insensitive).

```
get_uri(self)
```

Returns a SIPURI object which contains the adress, port and transport as parameter. This can be used to easily construct a RouteHeader:

```
route = Route("1.2.3.4", port=1234, transport="tls")
route_header = RouteHeader(route.get_uri())
```

SIP Accounts

Account Management is implemented in sipsimple/account.py (sipsimple.account module) and offers support for SIP accounts registered at SIP providers and SIP bonjour accounts which are discovered using mDNS.

AccountManager

The sipsimple.account.AccountManager is the entity responsible for loading and keeping track of the existing accounts. It is a singleton and can be instantiated anywhere, obtaining the same instance. It cannot be used until its start method has been called.

methods

```
__init__(self)
```

The __init__ method allows the AccountManager to be instantiated without passing any parameters. A reference to the AccountManager can be obtained anywhere before it is started.

```
start(self)
```

This method will load all the existing accounts from the configuration. If the Engine is running, the accounts will also activate. This method can only be called after the ConfigurationManager has been started. A SIPAccountManagerDidAddAccount will be sent for each account loaded. This method is called automatically by the SIPApplication when it initializes all the components of the middleware.

```
stop(self)
```

Calling this method will deactivate all accounts managed by the AccountManager. This method is called automatically by the SIPApplication when it stops.

```
has_account(self, id)
```

This method returns True if an account which has the specifed SIP ID (must be a string) exists and False otherwise.

```
get_account(self, id)
```

Returns the account (either an Account instance or the BonjourAccount instance) with the specified SIP ID. Will raise a KeyError if such an account does not exist.

```
get_accounts(self)
```

Returns a list containing all the managed accounts.

```
iter_accounts(self)
```

Returns an iterator through all the managed accounts.

```
find_account(self, contact_uri)
```

Returns an account with matches the specified contact_uri which must be a sipsimple.core.SIPURI instance. Only the accounts with the enabled flag set will be considered. Returns None if such an account does not exist.

notifications

```
SIPAccountManagerDidAddAccount
```

This notification is sent when a new account becomes available to the AccountManager. The notification is also sent when the accounts are loaded from the configuration.

timestamp:

A datetime.datetime object indicating when the notification was sent.

account:

The account object which was added.

```
SIPAccountManagerDidRemoveAccount
```

This notification is sent when an account is deleted using the delete method.

timestamp:

A datetime.datetime object indicating when the notification was sent.

account:

The account object which was deleted.

```
SIPAccountManagerDidChangeDefaultAccount
```

This notification is sent when the default account changes.

timestamp:

A datetime.datetime object indicating when the notification was sent.

old_account:

This is the account object which used to be the default account.

account:

This is the account object which is the new default account.

Account

The sipsimple.account.Account objects represent the SIP accounts which are registered at SIP providers. It has a dual purpose: it acts as both a container for account-related settings and as a complex object which can be used to interact with various per-account functions, such as presence, registration etc. This page documents the latter case, while the former is explained in the Configuration API.

There is exactly one instance of Account per SIP account used and it is uniquely identifiable by its SIP ID, in the form user@domain. It is a singleton, in the sense that instantiating Account using an already used SIP ID will return the same object. However, this is not the recommended way of accessing accounts, as this can lead to creation of new ones; the recommended way is by using the AccountManager. The next sections will use a lowercase, monospaced account to represent an instance of Account.

states

The Account objects have a setting flag called enabled which, if set to False will deactivate it: none of the internal functions will work in this case; in addition, the application using the middleware should not do anything with a disabled account. After changing it's value, the save() method needs to be called, as the flag is a setting and will not be used until this method is called:

```
account.enabled = True
account.save()
```

The Account objects will activate automatically when they are loaded/created if the enabled flag is set to True and the sipsimple.engine.Engine is running; if it is not running, the accounts will activate after the engine starts.

In order to create a new account, just create a new instance of Account with an id which doesn't belong to any other account.

The other functions of Account which run automatically have other enabled flags as well. They will only be activated when both the global enabled flag is set and the function-specific one. These are:

```
Account.registration.enabled
```

This flag controls the automatic registration of the account. The notifications SIPAccountRegistrationDidSucceed, SIPAccountRegistrationDidFail and SIPAccountRegistrationDidEnd are used to inform the status of this registration.

```
Account.presence.enabled
```

This flag controls the automatic subscription to buddies for the presence event and the publication of data in this event.

```
Account.dialog_event.enabled
```

This flag controls the automatic subscription to buddies for the dialog-info event and the publication of data in this event.

```
Account.message_summary.enabled
```

This flag controls the automatic subscription to the message-summary event in order to find out about voicemail messages.

The save() method needs to be called after changing these flags in order for them to take effect. The methods available on Account objects are inherited from SettingsObject.

attributes

The following attributes can be used on an Account object and need to be considered read-only.

```
id
```

This attribute is of type sipsimple.configuration.datatypes.SIPAddress (a subclass of str) and contains the SIP id of the account. It can be used as a normal string in the form user@domain, but it also allows access to the components via the attributes username and domain.

- account.id # 'alice@example.com'
- account.id.username # 'alice'
- account.id.domain # 'example.com'

```
contact
```

This attribute can be used to construct the Contact URI for SIP requests sent on behalf of this account. It's type is sipsimple.account.ContactURIFactory. It can be indexed by a string representing a transport ('udp', 'tcp', 'tls') or a sipsimple.util.Route object which will return a sipsimple.core.SIPURI object with the appropriate IP, port and transport. The username part is a randomly generated 8 character string consisting of lowercase letters; but it can be chosen by passing it to init when building the ContactURIFactory object.

- account.contact # 'ContactURIFactory(username=hnfkybrt)'
- account.contact.username # 'hnfkybrt'
- account.contact['udp'] # <SIPURI "sip:hnfkybrt@10.0.0.1:53024">
- account.contact['tls'] # <SIPURI "sip:hnfkybrt@10.0.0.1:54478;transport=tls">credentials

This attribute is of type sipsimple.core.Credentials which is built from the id.username attribute and the password setting of the Account. Whenever this setting is changed, this attribute is updated.

```
account.credentials # <Credentials for 'alice'>
```

```
uri
```

This attribute is of type sipsimple.core.SIPURI which can be used to form a FromHeader associated with this account. It contains the SIP ID of the account.

```
account.uri # <SIPURI "sip:alice@example.com">
```

notifications

```
CFGSettingsObjectDidChange
```

This notification is sent when the save() method is called on the account after some of the settings were changed. As the notification belongs to the SettingsObject class, it is exaplained in detail in SettingsObject Notifications.

```
SIPAccountDidActivate
```

This notification is sent when the Account activates. This can happen when the Account is loaded if it's enabled flag is set and the Engine is running, and at any later time when the status of the Engine changes or the enabled flag is modified.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPAccountDidDeactivate
```

This notification is sent when the Account deactivates. This can happend when the Engine is stopped or when the enabled flag of the account is set to False.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPAccountWillRegister
```

This notification is sent when the account is about to register for the first time. timestamp: A datetime.datetime object indicating when the notification was sent.

```
SIPAccountRegistrationWillRefresh
```

This notification is sent when a registration is about to be refreshed. timestamp: A datetime.datetime object indicating when the notification was sent.

```
SIPAccountRegistrationDidSucceed
```

This notification is sent when a REGISTER request sent for the account succeeds (it is also sent for each refresh of the registration). The data contained in this notification is:

timestamp:

A datetime.datetime object indicating when the notification was sent.

contact_header:

The Contact header which was registered.

contact_header_list:

A list containing all the contacts registered for this SIP account.

expires:

The amount in seconds in which this registration will expire.

registrar:

The sipsimple.util.Route object which was used.

```
SIPAccountRegistrationDidFail
```

This notification is sent when a REGISTER request sent for the account fails. It can fail either because a negative response was returned or because PJSIP considered the request failed (e.g. on timeout). The data contained in this notification is:

timestamp:

A datetime.datetime object indicating when the notification was sent.

error:

The reason for the failure of the REGISTER request.

timeout:

The amount in seconds as a float after which the registration will be tried again.

```
SIPAccountRegistrationDidEnd
```

This notification is sent when a registration is ended (the account is unregistered). The data contained in this notification is:

timestamp:

A datetime.datetime object indicating when the notification was sent.

registration:

The sipsimple.core.Registration object which ended.

```
SIPAccountRegistrationDidNotEnd
```

This notification is sent when a registration fails to end (the account is not unregistered). The data contained in this notification is:

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The SIP status code received.

reason:

The SIP status reason received.

registration:

The sipsimple.core.Registration object which ended.

```
SIPAccountRegistrationGotAnswer
```

This notification is sent whenever a response is received to a sent REGISTER request for this account. The data contained in this notification is:

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The SIP status code received.

reason:

The SIP status reason received.

registration:

The sipsimple.core.Registration object which was used.

registrar:

The sipsimple.util.Route object which was used.

```
SIPAccountMWIDidGetSummary
```

This notification is sent when a NOTIFY is received with a message-summary payload. The data contained in this notification is:

timestamp:

A datetime.datetime object indicating when the notification was sent.

message_summary:

A sipsimple.payloads.messagesummary.MessageSummary object with the parsed payload from the NOTIFY request.

BonjourAccount

The sipsimple.account.BonjourAccount represents the SIP account used for P2P mode; it does not interact with any server. The class is a singleton, as there can only be one such account on a system. Similar to the Account, it is used both as a complex object, which implements the functions for bonjour mode, as well as a container for the related settings.

states

The BonjourAccount has an enabled flag which controls whether this account will be used or not. If it is set to False, none of the internal functions will be activated and, in addition, the account should not be used by the application. The bonjour account can only activated if the Engine is running; once it is started, if the enabled flag is set, the account will activate. When the BonjourAccount is activated, it will broadcast the contact address on the LAN and discover its neighbours sending notifications as this happens.

attributes

The following attributes can be used on a BonjourAccount object and need to be considered read-only.

```
id
```

This attribute is of type sipsimple.configuration.datatypes.SIPAddress (a subclass of str) and contains the SIP id of the account, which is 'bonjour@local'. It can be used as a normal string, but it also allows access to the components via the attributes username and domain.

- bonjour_account.id # 'bonjour@local'
- bonjour_account.id.username # 'bonjour'
- bonjour_account.id.domain # 'local'

```
contact
```

This attribute can be used to construct the Contact URI for SIP requests sent on behalf of this account. It's type is sipsimple.account.ContactURIFactory. It can be indexed by a string representing a transport ('udp', 'tcp', 'tls') or a sipsimple.util.Route object which will return a sipsimple.core.SIPURI object with the appropriate IP, port and transport. The username part is a randomly generated 8 character string consisting of lowercase letters; but it can be chosen by passing it to init when building the ContactURIFactory object.

- bonjour_account.contact # 'ContactURIFactory(username=lxzvgack)'
- bonjour_account.contact.username # 'lxzvgack'
- bonjour_account.contact['udp'] # <SIPURI "sip:lxzvgack@10.0.0.1:53024">
- bonjour_account.contact['tls'] # <SIPURI "sip:lxzvgack@10.0.0.1:54478;transport=tls">credentials

This attribute is of type sipsimple.core.Credentials object which is built from the contact.username attribute; the password is set to the empty string.

- bonjour_account.credentials # <Credentials for 'alice'>

```
uri
```

This attribute is of type sipsimple.core.SIPURI which can be used to form a FromHeader associated with this account. It contains the contact address of the bonjour account:

- bonjour_account.uri # <SIPURI "sip:lxzvgack@10.0.0.1">

notifications

```
BonjourAccountDidAddNeighbour
```

This notification is sent when a new Bonjour neighbour is discovered.

service_description:

BonjourServiceDescription object uniquely identifying this neighbour in the mDNS library.

display_name:

The name of the neighbour as it is published.

host:

The hostname of the machine from which the Bonjour neighbour registered its contact address.

uri:

The contact URI of the Bonjour neighbour, as a FrozenSIPURI object.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountDidUpdateNeighbour
```

This notification is sent when an existing Bonjour neighbour has updates its published data.

service_description:

BonjourServiceDescription object uniquely identifying this neighbour in the mDNS library.

display_name:

The name of the neighbour as it is published.

host:

The hostname of the machine from which the Bonjour neighbour registered its contact address.

uri:

The contact URI of the Bonjour neighbour, as a FrozenSIPURI object.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountDidRemoveNeighbour
```

This notification is sent when a Bonjour neighbour unregisters.

service_description:

The BonjourServiceDescription object, which uniquely identifies a neighbour, that got unregistered.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountDiscoveryDidFail
```

This notification is sent once per transport when the Bonjour account has failed to perform the discovery process for the indicated transport.

reason:

String defining the reason of the failure.

transport:

String specifying the transport for which the discovery failed.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountDiscoveryFailure
```

This notification is sent once per transport when the Bonjour account has encountered a problem while browsing the list of neighbours for the indicated transport.

error:

String defining the error of the failure.

transport:

String specifying the transport for which the neighbour resoution failed.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountRegistrationDidEnd
```

This notification is sent once per transport when the Bonjour account unregisters its contact address for the indicated transport using mDNS.

transport:

String specifying the transport for which the registration ended.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountRegistrationDidFail
```

This notification is sent once per transport when the Bonjour account fails to register its contact address for the indicated transport using mDNS.

reason:

A human readable error message.

transport:

String specifying the transport for which the registration failed.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountRegistrationUpdateDidFail
```

This notification is sent once per transport when the Bonjour account fails to update its data for the indicated transport using mDNS.

reason:

A human readable error message.

transport:

String specifying the transport for which the registration update failed.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountRegistrationDidSucceed
```

This notification is sent once per transport when the Bonjour account successfully registers its contact address for the indicated transport using mDNS.

name:

The contact address registered.

transport:

String specifying the transport for which the registration succeeded.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountWillInitateDiscovery
```

This notification is sent when the Bonjour account is about to start the discovery process for the indicated transport.

transport:

String specifying the transport for which the discovery will be started.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
BonjourAccountWillRegister
```

This notification is sent just before the Bonjour account starts the registering process for the indicated transport.

transport:

String specifying the transport for which the registration will be started.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
CFGSettingsObjectDidChange
```

This notification is sent when the save() method is called on the account after some of the settings were changed. As the notification belongs to the SettingsObject class, it is explained in detail in SettingsObject Notifications.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPAccountWillActivate
```

This notification is sent when the Account is about to be activated, but before actually performing any activation task. See SIPAccountDidActivate for more detail.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPAccountDidActivate
```

This notification is sent when the BonjourAccount activates. This can happen when the BonjourAccount is loaded if it's enabled flag is set and the Engine is running, and at any later time when the status of the Engine changes or the enabled flag is modified.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPAccountWillDeactivate
```

This notification is sent when the Account is about to be deactivated, but before performing any deactivation task. See SIPAccountDidDeactivate for more detail.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPAccountDidDeactivate
```

This notification is sent when the BonjourAccount deactivates. This can happend when the Engine is stopped or when the enabled flag of the account is set to False.

timestamp:

A datetime.datetime object indicating when the notification was sent.

Audio API

The high-level audio API hides the complexity of using the low-level PJMEDIA interface. This is implemented in the sipsimple.audio module and contains the following components:

IaudioPort

an interface describing an object capable of producing and/or consuming audio data.

AudioDevice

an object conforming to the IAudioPort interface which describes a physical audio device.

AudioBridge

a collection of objects conforming to IAudioPort which connects all of them in a full mesh.

WavePlayer

an object conforming to the IAudioPort interface which can playback the audio data from a .wav file.

WaveRecorder

an object conforming to the IAudioPort interface which can record audio data to a .wav file.

IAudioPort

The IAudioPort interface describes an object capable of producing and/or consuming audio data. This can be a dynamic object, which changes its role during its lifetime and notifies such changes using a notification, which is part of the interface.

attributes

`mixer`

The AudioMixer this audio object is connected to. Only audio objects connected to the same mixer will be able to send audio data from one to another.

`consumer_slot`

An integer representing the slot (see AudioMixer) which this object uses to consume audio data, or None if this object is not a consumer.

`producer_slot`

An integer representing the slot (see AudioMixer) which this object uses to produce audio data, or None if this object is not a producer.

notifications

`AudioPortDidChangeSlots`

This notification needs to be sent by implementations of this interface when the slots it has change, so as to let the AudioBridges it is part of know that reconnections need to be made.

`consumer_slot_changed:`

A bool indicating whether the consumer slot was changed.

`producer_slot_changed:`

A bool indicating whether the producer slot was changed.

`old_consumer_slot:`

The old slot for consuming audio data. Only required if consumer_slot_changed is True.

`new_consumer_slot:`

The new slot for consuming audio data. Only required if consumer_slot_changed is True.

```
old_producer_slot:
```

The old slot for producing audio data. Only required if producer_slot_changed is True.

```
new_producer_slot:
```

The new slot for producing audio data. Only required if producer_slot_changed is True.

AudioDevice

The AudioDevice represents the physical audio device which is part of a AudioMixer, implementing the IAudioPort interface. As such, it can be uniquely identified by the mixer it represents.

methods

```
__init__(self, mixer, input_muted=False, output_muted=False):
```

Instantiates a new AudioDevice which represents the physical device associated with the specified AudioMixer. mixer: The AudioMixer whose physical device this object represents. input_muted: A boolean which indicates whether this object should act as a producer of audio data. output_muted: A boolean which indicates whether this object should act as a consumer of audio data.

attributes

```
input_muted
```

A writable property which controls whether this object should act as a producer of audio data. An AudioPortDidChange slots notification is sent when this attribute is changed to force connections to be reconsidered within the AudioBridges this object is part of.

```
output_muted
```

A writable property which controls whether this object should act as a consumer of audio data. An AudioPortDidChange slots notification is sent when this attribute is changed to force connections to be reconsidered within the AudioBridges this object is part of.

AudioBridge

The AudioBridge is the basic component which is able to connect IAudioPort implementations. It acts as a container which connects as the producers to all the consumers which are part of it. An object which is both a producer and a consumer of audio data will not be connected to itself. Being an implementation of IAudioPort itself, an AudioBridge can be part of another AudioBridge. The AudioBridge does not keep strong references to the ports it contains and once the port's reference count reaches 0, it is automatically removed from the AudioBridge.

Note: although this is not enforced, there should never be any cycles when connecting AudioBridges.

methods

```
__init__(self, mixer)
```

Instantiate a new AudioBridge which uses the specified AudioMixer for mixing.

```
add(self, port)
```

Add an implementation of IAudioPort to this AudioBridge. This will connect the new port to all the existing ports of the bridge. A port cannot be added more than once to an AudioBridge; thus, this object acts like a set.

```
remove(self, port)
```

Remove a port from this AudioBridge. The port must have previously been added to the AudioBridge, otherwise a ValueError is raised.

WavePlayer

A WavePlayer is an implementation of IAudioPort which is capable of producing audio data read from a .wav file. This object is completely reusable, as it can be started and stopped any number of times.

methods

```
__init__(self, mixer, filename, volume=100, loop_count=1,
pause_time=0, initial_play=True)
```

Instantiate a new WavePlayer which is capable of playing a .wav file repeatedly. All the parameters are available as attributes of the object, but should not be changed once the object has been started.

mixer:

The AudioMixer this object is connected to.

filename:

The full path to the .wav file from which audio data is to be read.

volume:

The volume at which the file should be played.

loop_count:

The number of times the file should be played, or 0 for infinity.

pause_time:

How many seconds to wait between successive plays of the file.

initial_play:

Whether or not the file to play once the WavePlayer is started, or to wait pause_time seconds before the first play.

```
start(self)
```

Start playing the .wav file.

```
stop(self)
```

Stop playing the .wav file immediately.

```
play(self)
```

Play the .wav file. This method is an alternative to the start/stop methods, it runs on a waitable green thread. One may call play().wait() in order to green-block waiting for the file playback to end.

attributes

`is_active`

A boolean indicating whether or not this WavePlayer is currently playing.

notifications

`WavePlayerDidStart`

This notification is sent when the WavePlayer starts playing the file the first time after the start() method has been called.

timestamp:

A datetime.datetime object indicating when the notification was sent.

`WavePlayerDidEnd`

This notification is sent when the WavePlayer is done playing either as a result of playing the number of times it was told to, or because the stop() method has been called.

timestamp:

A datetime.datetime object indicating when the notification was sent.

`WavePlayerDidFail`

This notification is sent when the WavePlayer is not capable of playing the .wav file.

timestamp:

A datetime.datetime object indicating when the notification was sent.

error:

The exception raised by the WaveFile which identifies the cause for not being able to play the .wav file.

WaveRecorder

A WaveRecorder is an implementation of IAudioPort is is capable of consuming audio data and writing it to a .wav file. Just like WavePlayer, this object is reusable: once stopped it can be started again, but if the filename attribute is not changed, the previously written file will be overwritten.

methods

```
__init__(self, mixer, filename)
```

Instantiate a new WaveRecorder.

mixer:

The AudioMixer this WaveRecorder is connected to.

filename:

The full path to the .wav file where this object should write the audio data. The file must be writable. The directories up to the file will be created if possible when the start() method is called.

```
start(self)
```

Start consuming audio data and writing it to the .wav file. If this object is not part of an AudioBridge, not audio data will be written.

```
stop(self)
```

Stop consuming audio data and close the .wav file.

attributes

```
is_active
```

A boolean indicating whether or not this WaveRecorder is currently recording audio data.

Conference

Conference support is implemented in the sipsimple.conference module. Currently, only audio conferencing is supported.

AudioConference

This class contains the basic implementation for audio conferencing. It acts as a container for AudioStream objects which it will connect in a full mesh, such that all participants can hear all other participants.

methods

```
__init__(self)
```

Instantiates a new AudioConference which is ready to contain AudioStream objects.

```
add(self, stream)
```

Add the specified AudioStream object to the conference.

```
remove(self, stream)
```

Removes the specified AudioStream object from the conference. Raises a ValueError if the stream is not part of the conference.

```
hold(self)
```

Puts the conference "on hold". This means that the audio device will be disconnected from the conference: all the participants will be able to continue the conference, but the local party will no longer contribute any audio data and will not receive any audio data using the input and output devices respectively. This does not affect the hold state of the streams in any way.

```
unhold(self)
```

Removes the conference "from hold". This means that the audio device will be reconnected to the conference: all the participants will start to hear the local party and the local party will start to hear all the participants. This does not affect the hold state of the streams in any way.

attributes

```
bridge
```

An AudioBridge which this conference uses to connect all audio streams. It can be used by the application to play a wav file using a WavePlayer to all the participants or record the whole conference using a WaveRecorder.

```
on_hold
```

A boolean indicating whether or not the conference is "on hold".

```
streams
```

The list of streams which are part of this conference. The application must not manipulate this list in any way.

XCAP API

The `sipsimple.xcap` module offers a high level API for managing XCAP resources to other parts of the middleware or to the applications built on top of the middleware. The XCAP resources which can be managed by means of this module are:

- **contact list**, by means of the `resource-lists` and `rls-services` XCAP applications
- **presence policies**, by means of the `org.openmobilealliance.pres-rules` or `pres-rules` XCAP applications
- **dialoginfo policies**, by means of the `org.openxcap.dialog-rules` XCAP application
- **status icon**, by means of the `org.openmobilealliance.pres-content` XCAP application
- **offline status**, by means of the `pidf-manipulation` XCAP application

The module can work with both OMA or IETF-compliant XCAP servers, preferring the OMA variants of the specification if the server supports them. Not all applications need to be available on the XCAP server, although it is obvious that only those that are will be managed. The central entity for XCAP resource management is the XCAPManager, whose API relies on a series of objects describing the resources stored on the XCAP server.

Contact

Implemented in sipsimple/xcap/__init__.py

A `Contact` is a URI with additional information stored about it, central to the XCAP contact list management. Information about a contact is stored in the `resource-lists`, `rls-services`, `org.openmobilealliance.pres-rules` or `pres-rules`, and `org.openxcap.dialog-rules` applications. The URI associated with the contact is considered a unique identifier. Information found in various places about the same URI is aggregated into a single `Contact` instance. More information about the contact is described within the attributes section.

attributes

`name`

A human-readable name which can be associated with the contact. This is stored using the `display-name` standard `resource-lists` element.

`uri`

The uniquely identifying URI.

`group`

A human-readable group name which can be used to group contacts together. If this is not `None`, the contact will be reachable from the `oma_buddylist` list within the `resource-lists` document, as defined by OMA. The group of a contact is the first `display-name` of an ancestor list which contains the contact information.

`subscribe_to_presence`

A boolean flag which indicates whether a subscription to the `presence` event is desired. If this is `True`, the contact's URI is referenced from a `rls-services` service which defines `presence` as one of the packages. Thus, a contact with this flag set is guaranteed to be referenced by an RLS service.

`subscribe_to_dialoginfo`

A boolean flag which indicates whether a subscription to the `dialog` event is desired. If this is `True`, the contact's URI is referenced from a `rls-services` service which defines `dialog` as one of the packages. Thus, a contact with this flag set is guaranteed to be refereneced by an RLS service.

`presence_policies`

Either `None` or a list of `PresencePolicy` objects which represent `org.openmobilealliance.pres-rules` or `pres-rules` rules which reference this contact's URI either directly (through an identity condition) or indirectly through resource lists (using the OMA external-list common policy extension).

`dialoginfo_policies`

Either `None` or a list of `DialoginfoPolicy` objects which represent `org.openxcap.dialog-rules` rules which reference this contact's URI through an identity condition.

`attributes`

A dictionary containing additional name, value pairs which the middleware or the application can use to store any information regarding this contact. This is stored through a proprietary AG-Projects extension to resource-lists.

methods

`__init__(self, name, uri, group, **attributes)`

Initializes a new `Contact` instance. The policies are by default set to `None` and the `subscribe_to_presence` and `subscribe_to_dialoginfo` flags to `True`.

Service

Implemented in sipsimple/xcap/__init__.py

A `Service` represents a URI managed by a Resource List Server (RLS). Subscriptions to this URI will be handled by the RLS.

attributes

`uri`

The URI which can be used to access a service provided by the RLS.

`packages`

A list of strings containing the SIP events which can be subscribed for to the URI.

`entries`

A list of URIs which represent the expanded list of URIs referenced by the service. A subscription to the service's URI for one of packages will result in the RLS subscribing to these URIs.

methods

```
__init__(self, uri, packages, entries=None)
```

Initializes a new `Service` instance.

Policy

Implemented in sipsimple/xcap/__init__.py

`Policy` is the base class for `PresencePolicy` and `DialoginfoPolicy`. It describes the attributes common to both.

attributes

```
id
```

A string containing the unique identifier of this specific policy. While it should not be considered human readable, OMA does assign specific meanings to some IDs.

```
action
```

A string having one of the values `"allow"`, `"confirm"`, `"polite-block"` or `"block"`.

```
validity
```

Either `None`, or a list of `datetime` instance 2-tuples representing the intervals when this policy applies. Example valid validity list which represents two intervals, each of two hours:

```
  from datetime import datetime, timedelta
  now = datetime.now()
  one_hour = timedelta(seconds=3600)
  one_day = timedelta(days=1)
  validity = [(now-one_hour, now+one_hour), (now+one_day-one_hour,
now+one_day+one_hour)]
```

```
sphere
```

Either `None` or a string representing the sphere of presentity when this policy applies.

```
multi_identity_conditions
```

Either `None` or a list of `CatchAllCondition` or `DomainCondition` objects as defined below. This is used to apply this policy to multiple users.

methods

```
__init__(self, id, action, name=None, validity=None, sphere=None,
multi_identity_conditions=None)
```

Initializes a new `Policy` instance.

```
check_validity(self, timestamp, sphere=Any)
```

Returns a boolean indicating whether this policy applies at the specific moment given by timestamp (which must be a `datetime` instance) in the context of the specific sphere.

CatchAllCondition

Implemented in sipsimple/xcap/__init__.py

`CatchAllCondition` represents a condition which matches any user, but which can have some exceptions.

attributes

```
exceptions
```

A list containing `DomainException` or `UserException` objects to define when this condition does not apply.

methods

```
__init__(self, exceptions=None)
```

Initializes a new `CatchAllCondition` instance.

DomainCondition

Implemented in sipsimple/xcap/__init__.py

`DomainCondition` represents a condition which matches any user within a specified domain, but which can have some exceptions.

attributes

`domain`

A string containing the domain for which this condition applies.

`exceptions`

A list containing `UserEception` objects to define when this condition does not apply.

methods

`__init__(self, domain, exceptions=None)`

Initializes a new `DomainCondition` instance.

DomainException

Implemented in sipsimple/xcap/__init__.py

`DomainException` is used as an exception for a `CatchAllCondition` which excludes all users within a specified domain.

attributes

```
domain
```

A string containing the domain which is to be excluded from the `CatchAllCondition` containing this object as an exception.

methods

```
__init__(self, domain)
```

Initializes a new `DomainException` instance.

UserException

Implemented in sipsimple/xcap/__init__.py

`UserException` is used as an exception for either a `CatchAllCondition` or a `DomainCondition` and excludes a user identified by an URI.

attributes

```
uri
```

A string containing the URI which is to be excluded from the `CatchAllCondition` or `DomainCondition` containing this object as an exception.

methods

```
__init__(self, uri)
```

Initializes a new `UserException` instance.

PresencePolicy

Implemented in sipsimple/xcap/__init__.py

A `PresencePolicy` represents either a `org.openmobilealliance.pres-rules` or `pres-rules` rule. It subclasses `Policy` and inherits its attributes, but defines additional attributes corresponding to the transformations which can be specified in a rule.

attributes

All of the following attributes only make sense for a policy having a `"allow"` action.

`provide_devices`

Either `sipsimple.util.All`, or a list of `Class`, `OccurenceID` or `DeviceID` objects as defined below.

`provide_persons`

Either `sipsimple.util.All`, or a list of `Class` or `OccurenceID` objects as defined below.

`provide_services`

Either sipsimple.util.All, or a list of Class, OccurenceID, ServiceURI or ServiceURIScheme objects as defined below.

`provide_activities`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_class`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_device_id`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_mood`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_place_is`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_place_type`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_privacy`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_relationship`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_status_icon`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_sphere`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_time_offset`

Either `None` (if the transformation is not be specified) or a boolean.

`provide_user_input`

Either `None` (if the transformation is not be specified) or one of the strings `"false"`, `"bare"`, `"thresholds"`, `"full"`.

`provide_unknown_attributes`

Either `None` (if the transformation is not be specified) or a boolean. The name of the attribute is not a typo, although it maps to the transformation named `provide-unknown-attribute` (singular form).

`provide_all_attributes`

Either `None` (if the transformation is not be specified) or a boolean.

methods

`__init__(self, id, action, name=None, validity=None, sphere=None, multi_identity_conditions=None)`

Initializes a new PresencePolicy instance. The provide_devices, provide_persons and provide_services are initialized to sipsimple.util.All, provide_all_attributes to True and the rest to None.

DialoginfoPolicy

Implemented in sipsimple/xcap/__init__.py

A `DialoginfoPolicy` represents a `org.openxcap.dialog-rules` rule. It subclasses `Policy` and inherits all of its attributes. It does not add any other attributes or methods and thus has an identical API.

Icon

Implemented in sipsimple/xcap/__init__.py

An `Icon` instance represents a status icon stored using the `org.openmobilealliance.pres-content` application.

attributes

`data`

The binary data of the image, as a string.

`mime_type`

The MIME type of the image, one of `image/jpeg`, `image/gif` or `image/png`.

`description`

An optional description of the icon.

`location`

An HTTP(S) URI which can be used by other users to download the status icon of the local user. If the XCAP server returns the proprietary X-AGP-Alternative-Location header in its GET and PUT responses, that is used otherwise the XCAP URI of the icon is used.

methods

`__init__(self, data, mime_type, description=None, location=None)`

Initializes a new `Icon` instance.

OfflineStatus

Implemented in sipsimple/xcap/__init__.py

An `OfflineStatus` instance represents data stored using the `pidf-manipulation` application.

attributes

```
activity
```

A string representing an activity within a `person` element.

```
note
```

A string stored as a note within a `person` element.

methods

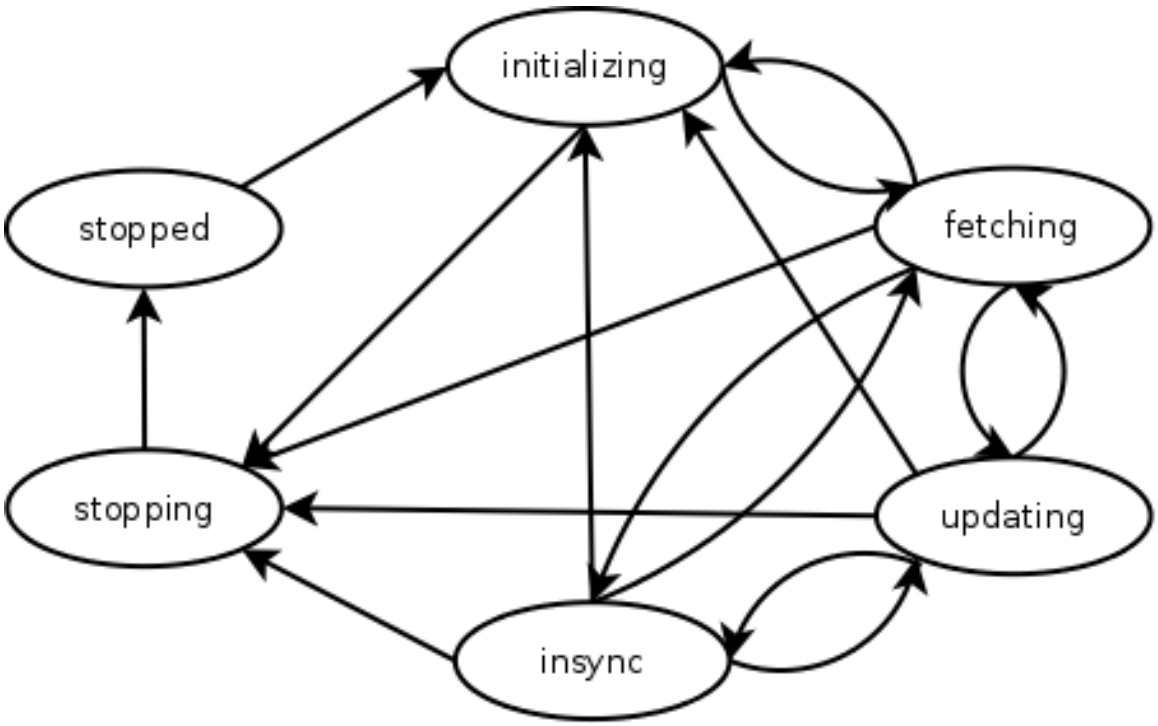
```
__init__(self, activity=None, note=Note)
```

Initializes a new `OfflineStatus` instance.

XCAPManager

Implemented in sipsimple/xcap/__init__.py

The XCAPManager is the central entity used to manage resource via the XCAP protocol. It uses a storage factory provided by SIPApplication through the Storage API. It has state machine as described in the following diagram:



The load method needs to be called just once in order to load the data from the cache. Once this is done, the start and stop methods can be called as needed. Initially in the stopped state, the start method will result in a transition to the initializing state. While in the initializing state, the XCAP manager will try to connect to the XCAP server and retrieve the capabilities (xcap-caps application). It will then initiate a SUBSCRIBE for the xcap-diff event (if configured) and transition to the fetching state. In the fetching state, it will try retrieve all the documents from the XCAP server, also specifying the ETag of the last known version. If none of the documents changed and this is not the first fetch, it transitions to the insync state. Otherwise, it inserts a normalize operation at the beginning of the journal (described below) and transitions to the updating state. In the updating state, it applies the operations from the journal which were not applied yet on the currently known documents and tries to push the documents, specifying the Etag of the last known version. If an operation fails due to a document having been modified, it marks all the operations in the journal as not being applied and transitions to the fetching state; if any other error occurs, the update is retried periodically. If the update succeeds, data is extracted from the documents and the XCAPManagerDidReloadData notification is sent. The XCAPManager then transitions to the insync state. A call to the stop method will result in a transition to the stopping state, termination of any existing SUBSCRIBE dialog and a transition to the stopped state.

Modifications to the settings which control the XCAPManager can result in either a transition to the `initializing` state or the termination of any previous SUBSCRIBE dialog and creation of a new one.

The subscription to the `xcap-diff` event allows the XCAPManager to be notified when the documents it manages are modified remotely. If the subscription fails, a fetch of all the documents is tried and the subscription is retried in some time. This allows the XCAPManager to reload the documents when they are modified remotely even if `xcap-diff` event is not supported by the provider. If subscription for xcap-diff event fails, a fetch of all the documents will be tried every 2 minutes.

The XCAPManager keeps the documents as they are stored on the XCAP server along with their ETags in an on-disk cache. All operations are made using the conditional `If-Match` and `If-None-Match` headers such that remote modifications the XCAPManager does not know about are not overwritten and bandwidth and processing power are not wasted by GET operations when a document is not modified.

Operations which the XCAPManager can be asked to apply to modify the documents are kept in a journal. This journal is saved to disk, such that operations which cannot be applied when requested due to server problems or lack of connectivity are retried even after application restarts. In addition, the high-level defined operations and the journal allow the modifications to be applied even if the document stored on the XCAP server are modified. Put differently, operations do depend on a specific version of the documents and the XCAPManager will try to apply them irrespective of the format of the document.

configuration

```
Account.id, Account.auth.username, Account.auth.password
```

These are used both for the `xcap-diff` subscription and the XCAP server connection.

```
Account.sip.subscribe_interval
```

This controls the Expires header used for the subscription, although a 423 response from the server can result in a larger Expires value being used.

```
Account.xcap.xcap_root
```

This specifies the XCAP root used for contacting the XCAP server and managing the resources. If this setting is `None`, a TXT DNS query is made for the `xcap` subdomain of the SIP account's domain. The result is interpreted as being an XCAP root. Example record for account alice@example.org:

```
xcap.example.org.    IN  TXT     "https://xcap.example.org/xcap-root"
```

```
SIPSimpleSettings.sip.transport_list
```

This controls the transports which can be used for the subscription.

methods

```
__init__(self, account)
```

Initializes an XCAPManager for the specified account.

```
load(self)
```

Allows the XCAPManager to the load its internal data from cache.

```
start(self)
```

Starts the XCAPManager. This will result in the subscription being started, the XCAP server being contacted and any operations in the journal being applied. This method must be called in a green thread.

```
stop(self)
```

Stops the XCAPManager from performing any tasks. Waits until the `xcap-diff` subscription, if any, is terminated. This method must be called in a green thread.

```
start_transaction(self)
```

This allows multiple operations to be queued and not applied immediately. All operations queued after a call to this method will not be applied until the `commit_transaction` method is called. This does not have the same meaning as a relational database transaction, since there is no `rollback` operation.

```
commit_transaction(self)
```

Applies the modifications queued after a call to `start_transaction`. This method needs to be called the exact same number of times the `start_transaction` method was called. Does not have any effect if `start_transaction` was not previously called.

The following methods results in XCAP operations being queued on the journal:

```
add_group(self, group)
```

Add a contact group with the specified name.

```
rename_group(self, old_name, new_name)
```

Change the name of the contact group `old_name` to `new_name`. If such a contact group does not exist, the operation does not do anything.

```
remove_group(self, group)
```

Remove the contact group (and any contacts contained in it) with the specified name. If such a contact group does not exist, the operation does not do anything.

```
add_contact(self, contact)
```

Adds a new contact, described by a `Contact` object. If the contact with the same URI and a not-`None` group already exists, the operation does not do anything. Otherwise, the contact is added and any reference to the contact's URI is overwritten. Requests to add a contact to some OMA reserved presence policies (`wp_prs_unlisted`, `wp_prs_allow_unlisted`, `wp_prs_block_anonymous`, `wp_prs_allow_own`) is ignored.

```
update_contact(self, contact, **attributes)
```

Modifies a contact's properties. The keywords can be any of the Contact attributes, with the same meaning. The URI of the contact to be modified is taken from the first argument. If such a URI does not exist, it is added. Requests to add a contact to some OMA reserved presence policies (wp_prs_unlisted, wp_prs_allow_unlisted, wp_prs_block_anonymous, wp_prs_allow_own) is ignored. The URI of a contact can be changed by specified the keyword argument uri with the new value.

```
remove_contact(self, contact)
```

Removes any reference to the contact's URI from all documents. This also means that the operation will make sure there are no policies which match the contact's URI.

```
add_presence_policy(self, policy)
```

Adds a new presence policy, described by a `PresencePolicy` object. If the id is specified and a policy with the same id exists, the operation does not do anything. Otherwise, if the id is not specified, one will be automatically generated (recommended). If the id is specified, but it is incompatible with the description of the policy (for example if an OMA defined id is used and there are some multi_identity_conditions), a new one will be automatically generated.

```
update_presence_policy(self, policy, **attributes)
```

Modifies a presence policy's properties. The keywords can be any of the `PresencePolicy` attributes, with the same meaning. The id of the policy to be modified is taken from the first argument. If such a policy does not exist and there is sufficient information about the policy, it is added. If the policy to be modified uses the OMA extension to reference resource-lists and multi_identity_conditions are specified in the keywords, a new policy whose properties are the combination of the existing policy and the keywords is created.

```
remove_presence_policy(self, policy)
```

Removes the presence policy identified by the id attribute of the `PresencePolicy` object specified. If the id is `None` or does not exist in the document, the operation does not do anything. Some standard OMA policies (`wp_prs_unlisted`, `wp_prs_allow_unlisted`, `wp_prs_block_anonymous`, `wp_prs_allow_own`, `wp_prs_grantedcontacts`, `wp_prs_blockedcontacts`) cannot be removed.

```
add_dialoginfo_policy(self, policy)
```

Adds a new dialoginfo policy, described by a DialoginfoPolicy object. If the id is specified and a policy with the same id exists, the operation does not do anything. Otherwise, if the id is not specified, one will be automatically generated (recommended).

```
update_dialoginfo_policy(self, policy, **attributes)
```

Modifies a dialoginfo policy's properties. The keywords can be any of the `DialoginfoPolicy` attributes, with the same meanining. The id of the policy to be modified is taken from the first argument. If such a policy does not exist and there is sufficient information about the policy, it is added.

```
remove_dialoginfo_policy(self, policy)
```

Removes the dialoginfo policy identified by the id attribute of the `DialoginfoPolicy` object specified. If the id is `None` or does not exist in the document, the operation does not do anything.

```
set_status_icon(self, icon)
```

Sets the status icon using the information from the `Icon` object specified. The `location` attribute is ignored. The MIME type must be one of `image/gif`, `image/png` or `image/jpeg`. If the argument is `None`, the status icon is deleted.

```
set_offline_status(self, status)
```

Sets the offline status using the information from the `OfflineStatus` object specified. If the argument is `None`, the offline status document is deleted.

notifications

```
XCAPManagerWillStart
```

This notification is sent just after calling the `start` method.

timestamp:

A `datetime.datetime` object indicating when the notification was sent.

```
XCAPManagerDidStart
```

This notification is sent after the XCAPManager has started doing its tasks (contacting the XCAP server, subscribing to `xcap-diff` event). This notification does not mean that any of these operations were successful, as the XCAPManager will retry them continuously should they fail.

timestamp:

A `datetime.datetime` object indicating when the notification was sent.

```
XCAPManagerWillEnd
```

This notification is sent just after calling the `stop` method.

timestamp:

A `datetime.datetime` object indicating when the notification was sent.

```
XCAPManagerDidEnd
```

This notification is sent when the XCAPManager has stopped performing any tasks. This also means that any active `xcap-diff` subscription has been terminated.

timestamp:

A `datetime.datetime` object indicating when the notification was sent.

```
XCAPManagerDidDiscoverServerCapabilities
```

This notification is sent when the XCAPManager contacts an XCAP server for the first time or after the connection is reset due to configuration changes. The data of the notification contains information about the server's capabilities (obtained using the `xcap-caps` application).

contactlist_supported:

A boolean indicating the support of documents needed for contact management (`resource-lists` and `rls-services`).

presence_policies_supported:

A boolean indicating the support of documents needed for presence policy management (`org.openmobilealliance.pres-rules` or `pres-rules`).

dialoginfo_policies_supported:

A boolean indicating the support of documents needed for dialoginfo policy management (`org.openxcap.dialog-rules`).

status_icon_supported:

A boolean indicating the support of documents needed for status icon management (`org.openmobilealliance.pres-content`).

offline_status_supported:

A boolean indicating the support of documents needed for offline status management (`pidf-manipulation`).

timestamp:

A `datetime.datetime` object indicating when the notification was sent.

```
XCAPManagerDidReloadData
```

This notification is sent when the XCAPManager synchronizes with the XCAP server. The data of the notification contains objects representing the data as it is stored on the XCAP server.

contacts:

A list of `Contact` objects.

groups:

A list of strings.

services:

A list of `Service` objects.

presence_policies:

A list of `PresencePolicy` objects.

dialoginfo_policies:

A list of `DialoginfoPolicy` objects.

status_icon:

A `StatusIcon` object if one is stored, `None` otherwise.

offline_status:

A `OfflineStatus` object if offline status information is stored, `None` otherwise.

timestamp:

A `datetime.datetime` object indicating when the notification was sent.

`XCAPManagerDidChangeState`

This notification is sent when the XCAPManager transitions from one state to another.

prev_state:

The old state of the XCAPManager, a string.

state:

The new state of the XCAPManager, a string.

timestamp:

A `datetime.datetime` object indicating when the notification was sent.

Threading API

The threading API is used by the middleware to provide an easy interface to create and use threads. Threads provided by this API are implemented using the EventQueue object which provides a easy way of serializing operations and running them in another thread. The API also provides decorators and functions to easily run functions in the desired thread.

Thread Manager

The Thread Manager is a Singleton responsible for creating and keeping track of EventQueue based threads. It's used by the configuration framework in order to run all configuration saving/deleting operations in a single thread, for example.

methods

```
__init__(self)
```

Instantiates a new ThreadManager or returns the already instantiated ThreadManager instance, since it's a Singleton.

```
start(self)
```

Start the ThreadManager. This method doesn't really do anything since threads are created by using the get_thread function.

```
stop(self)
```

Stop the ThreadManager. All threads will be stopped and joined.

```
get_thread(self, thread_id)
```

Return the thread matching the given ID. If the ID doesn't exist it will be created and added to the ThreadManager's internal data.

```
stop_thread(self, thread_id)
```

Stops the thread matching the given ID. The thread will also be joined from a internal thread whose ID is thread-ops.

utility functions and decorators

```
call_in_thread(thread_id, func, *args, **kwargs)
```

Function which will run the given func in the specified thread.

```
run_in_thread(thread_id)
```

Decorator which will run the decorated function in the specified thread.

The reactor thread and green threads

SIP SIMPLE SDK uses Twisted's reactor to run an event loop where some internal operations are performed.

Green threads are threads that work in userspace on top of one (or more) native threads in a cooperative way. They are not preemptive, so only one green thread may run at a given moment and another one can only run if the first yielded the control explicitly (unlike preemptive threads). Green threads run on top of the reactor thread.

utility functions and decorators

Utility functions and decorators are provided in order to run code in the reactor thread or in a new green thread in the sipsimple.threading and sipsimple.threading.green packages.

```
call_in_twisted_thread(func, *args, **kwargs)
```

Function which will run the given func in the reactor thread.

```
run_in_twisted_thread
```

Decorator which will run the decorated function in the reactor thread.

```
call_in_green_thread(func, *args, **kwargs)
```

Function which will spawn a new green thread and run the given func there.

```
run_in_green_thread
```

Decorator which will spawn a new green thread and run the decorated function there.

```
run_in_waitable_green_thread
```

Decorator which will spawn a new green thread and run the decorated function there with the ability of waiting for the result of the function. This will make code look blocking, but as it's running in a green thread it's not, since it's cooperating with other green threads.

Configuration API

The configuration API is used by the Middleware API to store and read its settings.

By managing the settings of the middleware through this configuration API one can create different applications that behave consistently and inherit the same settings. For example a command line client or a GUI program can read and write their settings through this API.

In addition, the configuration API offers an extensibility framework, by which applications can add their specific settings which will be managed in the same way as the middleware settings. The settings are loaded and saved from/to persistent storage using a backend; a backend is provided which has a simple text file format.

The settings can be managed by API calls. The middleware settings have appropriate defaults so that it can function properly with a minimum amount of changes.

Architecture

Configuration API consists of the low-level classes that can be used for storing and retrieving configuration objects. Moreover, it allows the creation of a higher level API for accessing configuration items. The SIP SIMPLE settings are defined using this API, however application-specific settings can also make use of it in order to define a consistent view of all the settings, by either extending the settings objects defined in the middleware or creating new settings objects.

The module sipsimple.configuration contains the main classes of the configuration API. These are:

1. ConfigurationManager
2. SettingsObject
3. SettingsGroup
4. Setting
5. SettingsObjectExtension

In addition, the exceptions which make up this package are:

1. ConfigurationError (base class for all other configuration errors)
2. ObjectNotFoundError

The package sipsimple.configuration.backend contains the abstract interface for configuration backends, as well as concrete implementations of backends. This package is explained in more detail in Configuration Backend API.

ConfigurationManager

The central entity is the ConfigurationManager, which is used for storing and retrieving settings. Within the ConfigurationManager, settings can be represented in a hierarchical structure, where the root the of tree is the configuration document. This structure is represented using a dictionary, defined recursively as:

1. the keys of the dictionary are unicode objects
2. the values of the dictionary can be:
3. the None object (this represents a default value)
4. unicode objects
5. lists of unicode objects
6. dictionaries using this specification

An item in the dictionary with an unicode object or a list as the value is a single setting: the name of the item is the name of the setting. An item with a dictionary as the value is a group of settings: the name of the item is the name of the group. This dictionary representation is stored to a persistent storage and loaded using the configuration backend as explained in Configuration Backend API. Any backend which is able to store and load such data can be used, but a simple text file backend is provided. After configuration data is loaded from the backend, it is saved on the ConfigurationManager and can be managed using its methods; in order to save the data using the backend provided, the save method needs to be called; any calls to update or delete will not ask the backend to store the data as well.

The ConfigurationManager class is a singleton to allow any part of the code to access it without the need to pass references. However, its start method needs to be called before it can be used. Once this is done, objects can be added, retrieved or deleted from the underlying storage; if using the SIPApplication class, its start method takes care of this passing as the backend the one it receives as an argument. The methods of ConfigurationManager are:

```
__init__(self)
```

References to the ConfigurationManager instance can be obtained anytime without passing any arguments. However, it needs the manager needs to be started in order to be used, otherwise all methods will raise a RuntimeError.

```
start(self, backend)
```

The start method allows the ConfigurationManager instance to use the specified backend for accessing the underlying storage. See Configuration Backend API for information on what the required interface for the passed object is. Raises a ConfigurationBackendError if the backend cannot load the configuration data from persistent storage.

```
update(self, group, name, data)
```

The partial data which must be a dictionary as formerly defined corresponding to an object having the specified name under the specified group. If group is None, the object will be saved top-level (its name will the a top-level key in the data

dictionary). Note that changes will not be written to the underlying storage until the save method is called.

```
delete(self, group, name)
```

If an object stored as name exists in group, then it will be deleted. If group is None, then the top-level object identified by name will be deleted.

```
get(self, group, name)
```

Retrieves the object stored with name in group, if it exists. Otherwise, the method will raise an ObjectNotFoundError. If group is None, the top-level object identified by name will be retrieved.

```
get_names(self, group)
```

Returns a list with all the names of the objects in group. Returns an empty list if the group does not exist.

```
save(self)
```

Flushes the changes made to the configuration to the backend. This method must be called to ensure that all changes have been written to persistent storage. Raises a ConfigurationBackendError if the backend cannot store the data to persistent storage.

SettingsObject

A SettingsObject is used to represent a hierarchy of settings, which are managed via the ConfigurationManager. There are two types of SettingsObject:

1. SettingsObjects without an ID, i.e. there should be only one instance of this SettingsObject in an application. The application should define these objects using the Singleton metaclass. This also means that the object cannot be deleted. An example of such a SettingsObject is SIPSimpleSettings. These SettingsObjects are useful to represent global settings.
2. SettingsObjects with an associated id. The instances are not necessarily persistent. New ones can be created and existing ones can be deleted. An example of such a SettingsObject is the Account. These SettingsObjects are useful to represent settings which apply to entities identifiable by a string id. The ID can be changed, which means these objects cannot be Singletons.

SettingsObjects can belong to a group, depending on whether the __group__ attribute was specified. If it wasn't, the data will be saved top-level using the id of the SettingsObject; otherwise, the data will be saved under the group specified using the id. It is recommended that SettingsObjects with instances per id be saved in a group, although this is not enforced. For example, the Account instances are saved in a group named Accounts.

When a SettingsObject is instantiated its contained settings are loaded from the configuration storage. If it is the first time a SettingsObject is created, the default values for the settings will apply. The SettingsObject will only be copied to storage when its save method is called.

Defining a global SettingsObject

In order to define a global SettingsObject, the __id__ attribute must be defined on the class, while the __group__ attribute can be defined. The __id__ must not be used in any other SettingsObject which is stored in the same group (or globally if the __group__ attribute is missing).

An example of defining a global SettingsObject:

```
from application.python.util import Singleton
from sipsimple.configuration import SettingsObject

class SIPSimpleSettings(SettingsObject):
    __metaclass__ = Singleton
    __group__ = 'Global'
    __id__ = 'SIPSimple'
```

The __init__ method must not accept any other argument except self. It will be called each time the settings are loaded from the storage, not only the first time the object is created.

Defining a per-id SettingsObject

In order to define a per-id SettingsObject, the __group__ attribute should be defined on the class, while the __id__ attribute must be left to None. When instantiating the resulting class, exactly one argument must be given, which represents the string id.

Each class defined as a per-id SettingsObject should be allocated a different group from all the other SettingsObjects (including global ones), otherwise the keys under which the SettingsObjects are stored could overlap. An example of defining a per-id SettingsObject:

```
from sipsimple.configuration import SettingsObject

class Account(SettingsObject):
    __group__ = 'Accounts'
    def __init__(self, id):
        """Do something each time the Account is loaded"""
```

The __init__ method must accept exactly one argument except self. It will be called each time the object is loaded from the storage, in addition to the first time the object is created. This allows the SettingsObject to be more than a simple collection of settings.

methods

```
save(self)
```

If the contained Settings of this SettingsObject have changed, the object will be saved to the persistent storage. A CFGSettingsObjectDidChange notification will be issued which contains the modified settings. If the save fails, a CFGManagerSaveFailed notification is issued in addition.

```
delete(self)
```

This method can only be called on per-id SettingsObjects. It will delete the object from the persistent storage. All references to this SettingsObject must be removed.

Notifications

```
CFGManagerSaveFailed
```

This notification is sent when the save or delete method of a SettingsObject fail to save the data in the configuration manager storage backend. Attributes:

modified:

A dict instance which maps settings keys in their fully qualified form (attribute names separated by '.', relative to the SettingsObject) to a ModifiedValue instance; the ModifiedValue instance contains two attributes: old and new which are set to the old and the new Setting's value respectively. If the object was never saved before, it will be None.

exception:

The exception object that was raised while attempting to save the data in the configuration manager.

operation:

Attempted opperation, one of 'save' or 'delete'.

timestamp:

A datetime.datetime object representing the moment the notification was sent.

```
CFGSettingsObjectDidChange
```

This notification is sent when the save method of a SettingsObject is called and some settings were previously changed. Attributes:

modified:

A dict instance which maps settings keys in their fully qualified form (attribute names seperated by '.', relative to the SettingsObject) to a ModifiedValue instance; the ModifiedValue instance contains two attributes: old and new which are set to the old and the new Setting's value respectively.

timestamp:

A datetime.datetime object representing the moment the notification was sent.

```
CFGSettingsObjectDidChangeID
```

This notification is sent when the save method of SettingsObject is called and the ID of the object was previously changed. Attributes:

old_id:

```
The old value of the ID.
```

new_id:

The new value of the ID.

timestamp:

A datetime.datetime object representing the moment the notification was sent.

Setting

The Setting descriptor is used to describe a setting in SettingsObjects. The following methods are part of the public API of it:

```
__init__(self, type, default=None, nillable=False)
```

Create a new Setting descriptor which represents a setting in all instances of a SettingsObject. The default value must be specified if the setting is not nillable. The type will be applied to the values which are set to this descriptor if the value is not already an instance of the type; it is not applied to the default value.

An example of using a setting:

```
from application.python.util import Singleton
from sipsimple import __version__
from sipsimple.configuration import Setting, SettingsObject

class SIPSimpleSettings(SettingsObject):
    __metaclass__ = Singleton
    __group__ = 'Global'
    __id__ = 'SIPSimple'

    user_agent = Setting(type=str, default='sipsimple %s' %
__version__)
```

When a setting value is read from the configuration backend, the type is used to reconstruct the value from a unicode object, a list of unicode objects, or a dictionary containing unicode keys and values with any of these three types. Several built-in types are recognised and are handled automatically:

1. bool: the unicode strings u'yes', u'true', {{{u'on' and u'1' are considered to have the value True, while u'no', u'false', u'off' and u'0' are considered to have the value False; the comparison is done case insensitively; all other strings are considered invalid.
2. int, long and basestring: the type is called using the value as an argument.

All other types are instantiated using an un-pickling like mechanism. The __new__ method is called without any arguments and __setstate__ is called on the object returned by __new__ using the value as the sole argument.

Saving a setting value is done similarly, according to type. The builtin types which are handled are the same:

1. bool: the unicode objects u'true' and u'false are used depending on the value.
2. int, long and basestring: unicode is called with the value as the sole argument.
3. For all other types, the __getstate__ method is called which should return an appropriate value.

SettingsGroup

A SettingsGroup allows settings to be structured hierarchically. Subclasses of SettingsGroup are containers for Settings and other SettingsGroups just as SettingsObjects are. In addition, the subclasses of SettingsGroup are descriptors and can be used as such to assign a SettingsGroup as a child of another SettingsGroup or a SettingsObject. An example usage containing Setting, SettingsGroup and SettingsObject:

```
from application.python.util import Singleton
from sipsimple import __version__
from sipsimple.configuration import Setting, SettingsGroup,
SettingsObject

class TLSSettings(SettingsGroup):
    verify_server = Setting(type=bool, default=False)

class SIPSimpleSettings(SettingsObject):
    __metaclass__ = Singleton
    __group__ = 'Global'
    __id__ = 'SIPSimple'

    user_agent = Setting(type=str, default='sipsimple %s' %
__version__)

    tls = TLSSettings
```

SettingsObjectExtension

The SettingsObjectExtension allows an application to add or customize the settings of the middleware according to its needs. In order to add or replace settings/settings groups defined in another SettingsObject, SettingsObjectExtension can be subclassed and the register_extension class method of the original SettingsObject can be called passing the SettingObjectExtension subclass as the sole argument. In order to add/replace settings in a group of settings, the original SettingsGroup can be subclassed. Example:

```
from sipsimple import __version__
from sipsimple.configuration import Setting, SettingsGroup,
SettingsObject, SettingsObjectExtension

class TLSSettings(SettingsGroup):
    verify_server = Setting(type=bool, default=False)

class SIPSimpleSettings(SettingsObject):
    __group__ = 'Global'
    __id__ = 'SIPSimple'

    user_agent = Setting(type=str, default='sipsimple %s' %
__version__)

    tls = TLSSettings

class TLSSettingsExtension(TLSSettings):
    verify_client = Setting(type=bool, default=True)

class SIPSimpleSettingsExtension(SettingsObjectExtension):
    default_account = Setting(type=str, default=None, nillable=True)

    tls = TLSSettingsExtension

SIPSimpleSettings.register_extension(SIPSimpleSettingsExtension)
```

Backend API

The backend API provides a way to use the configuration framework consistently, while using any system for storing the data persistently. The ConfigurationManager makes use of a backend whenever it needs to write/read something to the persistent storage. The backend only needs to know how to handle data in the dictionary format explained in Configuration Manager. In order to use a specific backend, it is given to the ConfigurationManager in its start method.

The interface sipsimple.configuration.backend.IBackend describes the backend:

```
load()
```

Load the configuration data using whatever means employed by the backend implementation and return a dictionary conforming to the definition in Configuration Manager.

```
save(data)
```

Given a dictionary conforming to the definition in this interface, save the data using whatever means employed by the backend implementation.

FileBackend

A concrete implementation of the IBackend interface resides in sipsimple.configuration.backend.file.FileBackend. The methods different from the ones in IBackend are:

```
__init__(self, filename, encoding='utf-8')
```

Create a new FileBackend which uses the specified filename for loading and storing the data to; the data is written using the specified encoding, defaulting to UTF-8.

This object saves the data using a simple text file format with the following syntax:

SettingGroups, SettingsObjects or Groups of SettingsObjects are represented by their name (or id in the case of SettingsObjects) followed by a colon (:). These containers can contain other such containers or simple settings. Their children need to be indented more that the container itself. The indentation need not be consistent.

```
Accounts:
    user@domain:
      display_name = User
      tls:
        certificate =
```

Simple settings are represented by a name followed by an equals sign and the value; whitespace anywhere in between is ignored. The different values are represented in the following way:

None is represented by the absence of a value.

```
setting =
```

Unicode objects are represented by a simple string (which can be quoted to include leading and trailing whitespace by either single or double quotes) and can have the following espace sequances: \' , \", \n, \r. The unicode characters are encoded using the encoding specified in the constructor.

```
    setting1 = value
    setting2 = value with spaces
    setting3 = "  value with leading and trailing spaces  "
    setting4 = value with a line feed\n
```

Lists are represented by unicode strings as described above separated by commas (,). Any not-quoted whitespace around the comma is ignored.

```
setting = a, b  , c
```

Complex settings can be represented just like a group:

```
complex_setting:
      field1 = value
      field2 = 123
```

Middleware Settings

These are the current settings, kept in the modules sipsimple.configuration.settings and sipsimple.account. The main classes used to access the settings are Account, BonjourAccount and SIPSimpleSettings. All settings can be accessed as simple attributes. The types of attributes is described for each setting below. When setting the value of an attribute, if it's not of the required type, it will be given to the specified type as the only argument. The modified settings are not saved to the persistent storage until the save method is called on the main object. Once this is done, a CFGSettingsObjectDidChange notification is sent, the data of which is explained in SettingsObject Notifications.

Only a nillable setting can be assigned the value None, even if the type of the setting would otherwise accept None as an argument. The settings as described below are not nillable, unless specified explicitely. To reset the value of a setting, the special object sipsimple.configuration.DefaultValue can be assigned to it. If a default value is not explicitely specified below, it defaults to None. Note that a non-nillable setting cannot have the default value of None.

General

```
SIP SIMPLE settings:
             +-- default_account = user@example.com
SIP SIMPLE --|-- user_agent = sipsimple
             |-- audio
             |-- chat
             |-- desktop_sharing
             |-- file_transfer
             |-- logs
             |-- msrp
             |-- rtp
             |-- sip
             +-- tls

        +-- alert_device = system_default
audio --|-- input_device = system_default
        |-- output_device = system_default
        |-- sample_rate = 44100
        |-- silent = False
        +-- tail_length = 200

       +
chat --|
       +

                   +
desktop_sharing --|
                   +

                  +
file_transfer --|
                  +

       +-- pjsip_level = 5
```

```
logs --|
       +

       +-- transport = tls
msrp --|
       +

      +-- audio_codec_list = AudioCodecList(['speex', 'G722', 'PCMU',
'PCMA', 'iLBC', 'GSM'])
rtp --|-- port_range = PortRange(start=50000, end=50400)
      +-- timeout = 30

      +-- tcp_port = 0
sip --|-- tls_port = 0
      |-- transport_list = SIPTransportList(['tls', 'tcp', 'udp'])
      +-- udp_port = 0

      +-- ca_list = None
tls --|-- protocol = TLSv1
      +-- timeout = 1000
```

The sipsimple.configuration.settings.SIPSimpleSettings class is a singleton can be instantiated and used anywhere after the ConfigurationManager has been started.

The settings are explained below:

```
SIPSimpleSettings.default_account (type=str,
default='bonjour@local', nillable=True)
```

A string, which contains the id of the default Account. This setting is managed by the AccountManager and should not be changed manually. See AccountManager for more information.

```
SIPSimpleSettings.user_agent (type=str, default='sipsimple VERSION')
```

This setting will be used to set the value of the User-Agent header in outgoing SIP requests and of the Server header in all SIP responses.

Audio

```
SIPSimpleSettings.audio.input_device (type=AudioInputDevice,
default='system_default', nillable=True)
```

The name of the audio device, which will be used for input (recording). If it is set to 'system_default', one will be selected automatically by the operating system; if it is set to None, a dummy device will be used which doesn't record anything.

```
SIPSimpleSettings.audio.output_device (type=AudioOutputDevice,
default='system_default', nillable=True)
```

The name of the audio device, which will be used for output (playback). If it is set to 'system_default'}}, one will be selected automatically by the operating system; if it is set to {{{None, a dummy device will be used which will discard any audio data.

```
SIPSimpleSettings.audio.alert_device (type=AudioOutputDevice,
default='system_default', nillable=True)
```

The name of the alert device, which can be used for alerting the user. If it is set to 'system_default', one will be selected automatically by the operating system; if it is set to None, a dummy device will be used which will discard any audio data. This device is not used by the middleware but is provided for consistency.

```
SIPSimpleSettings.audio.tail_length (type=NonNegativeInteger,
default=200)
```

This setting is used as a parameter for the audio echo cancellation algorithm. It's value is a non-negative integer which represents milliseconds. It specifies the length of the echo cancellation filter.

```
SIPSimpleSettings.audio.sample_rate (type=SampleRate, default=16000)
```

This is the sample rate at which the audio system runs, in Hz. All playback and recording will be done at this rate. If an audio codec has a smaller or larger sample rate, it will be resampled to this value (if possible). Example values include 8000, 32000, 44100 etc.

```
SIPSimpleSettings.audio.silent (type=bool, default=False)
```

If this setting is set to True, no audio notifications will be played on the alert device (the volume of the alert device will be set to 0).

Chat

Empty section for future use.

Desktop Sharing

Empty section for future use.

File Transfer

Empty section for future use.

Logs

```
SIPSimpleSettings.logs.pjsip_level (type=NonNegativeInteger, default=5)
```

This setting controls the amount of log messages generated by the PJSIP core. It must be set to a non-negative integer.

MSRP

```
SIPSimpleSettings.msrp.transport (type=MSRPTransport, default='tls')
```

MSRP can use either TLS or TCP and this setting controls which one should be used for normal accounts.

RTP

```
SIPSimpleSettings.rtp.port_range (type=PortRange,
default=PortRange(50000, 50400))
```

This setting controls the port range from which ports used by RTP transport will be assigned. The values of the ports need to be in the range 1-65535; the start port must not be larger than the end port.

```
SIPSimpleSettings.rtp.audio_codec_list (type=AudioCodecLis}t}},
default={{{AudioCodecList(('speex', 'G722', 'PCMU', 'PCMA')))
```

This setting is used to specify the preferred audio codecs, which should be used for audio calls. It must contain only strings, which represent the supported codecs (speex, G722, PCMA, PCMU, iLBC and GSM), in the order in which they are preferred. This setting can be overridden per account.

SIP

```
SIPSimpleSettings.sip.udp_port (type=Port, default=0)
```

This is the port on which the Engine will bind and for for sending and receiving UDP packets. It is an integer in the range 0-65535. If it is set to 0, it will be allocated automatically.

```
SIPSimpleSettings.sip.tcp_port (type=Port, default=0)
```

This is the port on which the Engine will listen for TCP connections. It is an integer in the range 0-65535. If it is set to 0, it will be allocated automatically.

```
SIPSimpleSettings.sip.tls_port (type=Port, default=0)
```

This is the port on which the Engine will listen for TLS connections. It is an integer in the range 0-65535. If it is set to 0, it will be allocated automatically. The port must be different than the port used for TCP connections.

```
SIPSimpleSettings.sip.transport_list (type=SIPTransportList,
default=SIPTransportList(('tls', 'tcp', 'udp')))
```

This setting's value is a tuple, which can only contain the strings 'tls', 'tcp' and 'udp'. It has a double purpose:

Only the transports specified here are used to SIP requests associated with normal accounts.

The order of the transports specified in this tuple represent the preferred order in which transports should be used. This applies to all SIP requests.

TLS

```
SIPSimpleSettings.tls.ca_list (type=Path, default=None, nillable=True)
```

The settings points to a file which contains the CA certificates. In can be None, in which case no CAs are available. It is interpreted as an absolute path, with a leading

~ expanded to the home directory of the current user. In order to access the full path to the CA file, the normalized attribute on the setting can be used:

```
SIPSimpleSettings().tls.ca_list.normalized
```

SIPSimpleSettings.tls.protocol (type=TLSProtocol, default='TLSv1')

This setting sets the version of the TLS protocol which will be used. It is a string and must be one of 'TLSv1'.

```
SIPSimpleSettings.tls.timeout (type=NonNegativeInteger, default=1000)
```

This is the timeout for negotiating TLS connections, in milliseconds. It must be an non-negative integer.

Account

```
Account user@example.com:
          +-- display_name = Example User
account --|-- enabled = True
          |-- auth
          |-- dialog_event
          |-- message_summary
          |-- nat_traversal
          |-- presence
          |-- pstn
          |-- rtp
          |-- sip
          |-- tls
          +-- xcap

        +-- password = xyz
auth --|-- username = None
        +

               +-- enabled = True
dialog_event --|
               +

                  +-- enabled = True
message_summary --|-- voicemail_uri = None
                  +

                +-- msrp_relay = None
nat_traversal --|-- stun_server_list = None
                |-- use_ice = False
                |-- use_msrp_relay_for_inbound = True
                +-- use_msrp_relay_for_outbound = False

           +-- enabled = True
presence --|-- use_rls = True
           +

       +
pstn --|
       +
```

```
      +-- audio_codec_list = None
rtp --|-- inband_dtmf = False
      |-- srtp_encryption = optional
      +-- use_srtp_without_tls = False

      +-- outbound_proxy = SIPProxyAddress('sip.example.com',
port=5060, transport='udp')
sip --|-- publish_interval = 3600
      |-- always_use_my_proxy = False
      |-- register = True
      |-- register_interval = 600
      +-- subscribe_interval = 3600


      +-- certificate = tls/user@example.com.crt
tls --|-- verify_server = False
      +

       +-- enabled = True
xcap --|
       +-- xcap_root = https://xcap.example.com/xcap-root/
```

The Account object is used to represent a normal SIP account registered at a SIP provider. It is uniquely identifiable by it's SIP ID, in the form user@domain. There is exactly one instance of Account per ID, which means that an Account can be accessed by instantianting it anywhere. However, this is not the recommended way of accessing accounts, since it can lead to creating new accounts. The recommended way is by using the AccountManager. Information about the roles of Account, apart from being a collection of settings, is explained in the Middleware API.

The settings that can be accessed on an Account are described below:

```
Account.id (type=SIPAddress)
```

Its type is a subclass of str, so it can be used as a normal string, however it also has two attributes username and domain which point to the specific parts of the SIP address.

```
Account.display_name (type=str, default=None, nillable=True)
```

The contents of this setting will be sent as part of the From header when sending SIP requests, the From CPIM header and other similar information.

```
Account.enabled (type=bool, default=False)
```

If this setting is set to True, the Account will automatically activate and can be used in other parts of the middleware. More about this is described in Account.

Auth

```
Account.auth.username (type=str, default=None, nillable=True)
```

The username used for authentication if it is different from the one in the SIP ID of the account. If it is None or an empty string, the account SIP ID username will be used instead.

```
Account.auth.password (type=str, default='')
```

The password, which will be used with this account for authentication.

Dialog Event

```
Account.dialog_event.enabled (type=bool, default=True)
```

If this setting is set to True, the Account will subscribe to the dialog event as specified by RFC4235.

Message Summary

```
Account.message_summary.enabled (type=bool, default=True)
```

If this setting is set to True, the Account will subscribe to the message-summary event, as specified by RFC3842.

```
Account.message_summary.voicemail_uri (type=str, default=None,
nillable=True)
```

This is the SIP URI where the Subscribe for message-summary is sent.

NAT Traversal

```
Account.nat_traversal.use_ice (type=bool, default=False)
```

If this setting is set to True, ICE will be used for finding media candidates for communication over NAT-ed networks.

```
Account.nat_traversal.stun_server_list (type=StunServerAddressList,
default=None, nillable=True)
```

This setting used for NAT traversal can be used to specify the addresses of the STUN servers used for detecting server reflexive candidates in the context of ICE. The value of the setting is a tuple of objects of type StunServerAddress. If None, the servers will be looked up in the DNS (SRV record _stun._udp.domain).

```
Account.nat_traversal.msrp_relay (type=MSRPRelayAddress, default=None,
nillable=True)
```

This setting can be used to specify a MSRP relay for use in MSRP connections. If it is set to None. If None, the servers will be looked up in the DNS (SRV record _msrps._tcp.domain).

```
Account.nat_traversal.use_msrp_relay_for_inbound (type=bool,
default=True)
```

If this setting is set to True, the MSRP relay will be used for all incoming MSRP connections.

```
Account.nat_traversal.use_msrp_relay_for_outbound (type=bool,
default=False)
```

If this setting is set to True, the MSRP relay will be used for all outgoing MSRP connections.

Presence

```
Account.presence.enabled (type=bool, default=True)
```

If this setting is set to True, the Account will publish its presence state and subscribe to presence and presence.winfo Event packages.

```
Account.presence.use_rls (type=bool, default=False)
```

If this setting is set to True, the Account will store its Buddy Lists in rls-services XCAP document and send a single Subscribe for the presence event to the RLS services address to obtain the presence information for its buddies. If it is set to False, it will subscribe to each buddy individually.

RTP

```
Account.rtp.audio_codecs (type=AudioCodecList, default=None,
nillable=True)
```

This setting is used to specify the preferred audio codecs, which should be used for audio calls of this account. It must be a tuple containing only strings, which represent the supported codecs (speex, g722, g711, ilbc and gsm), in the order in which they are preferred, or None if the codec_list from the general rtp settings is to be used.

```
Account.audio.srtp_encryption (type=SRTPEncryption, default='optional')
```

The value of this setting specifies how the account requires the calls to be encrypted using SRTP. It can be one of the values 'disabled', 'optional' or 'mandatory'.

```
Account.audio.use_srtp_without_tls (type=bool, default=False)
```

If this setting is set to True, SRTP could be used even if the SIP signaling used to control the call is not over TLS.

SIP

```
Account.sip.always_use_my_proxy (type=bool, default=False)
```

If this setting is set to True and the outbound proxy is not set, the signalling for outbound requests going to foreign domains will be sent to the account proxy instead of sending it to the foreign proxy.

```
Account.sip.outbound_proxy (type=SIPProxyAddress, default=None,
nillable=True)
```

This setting specifies whether to send all SIP requests when creating a new SIP dialog to a specific proxy. If this setting is set to None, then an RFC3263 lookup will be done based on the domain part of the SIP request URI.

```
Account.sip.register (type=bool, default=True)
```

If this setting is set to True, the Account will automatically register when it is active. More about this is described in Account.

```
Account.sip.publish_interval (type=NonNegativeInteger, default=600)
```

This setting controls the number of seconds used for the Expire header when publishing events. It must be a non-negative integer.

```
Account.sip.subscribe_interval (type=NonNegativeInteger, default=600)
```

This setting controls the number of seconds used for the Expire header when subscribing to events. It must be a non-negative integer.

```
Account.registration.interval (type=NonNegativeInteger, default=600)
```

This setting controls the number of seconds used for the Expire header when registering. It must be a non-negative integer.

TLS

```
Account.tls.certificate (type=Path, default=None, nillable=True)
```

The path to the file that contains the certificate and its private key used to authenticate on TLS connections. It is interpreted as an absolute path, with a leading ~ expanded to the home directory of the current user. In order to access the full path to the TLS certificate, the normalized attribute on the setting can be used:

```
account.tls.certificate.normalized
Account.tls.verify_server (type=bool, default=False)
```

If this setting is set to True, the middleware will verify the server's certificate when connecting via TLS.

XCAP

```
Account.xcap.enabled (type=bool, default=True)
```

If this setting is set to True, The use of XCAP root set below will be activated.

Account.xcap.xcap_root (type=XCAPRoot, default=None, nillable=True)

The XCAP root is required for accessing documents via the XCAP protocol. It must be a URL with either the http or https schemes.

```
Account.xcap.use_xcap_diff (type=bool, default=True)
```

If this setting is set to True, the Account will subscribe to the xcap-diff event in order to find out if the XCAP documents handled by the Account are modified by another entity.

BonjourAccount

```
Account bonjour@local:
          +-- display_name = Bonjour User
account --|-- enabled = False
          |-- msrp
          |-- rtp
          |-- sip
          +-- tls

       +-- transport = tcp
msrp --|
       +

      +-- audio_codec_list = None
rtp --|-- inband_dtmf = False
      |-- srtp_encryption = optional
      +-- use_srtp_without_tls = False

      +-- transport_list = SIPTransportList(['udp'])
sip --|
      +

      +-- certificate = tls/bonjour@local.crt
tls --|-- verify_server = False
      +
```

The BonjourAccount is a singleton object as there can only be one bonjour account on a system. A bonjour account is used in P2P mode and does not interact with any server. Similar to the Account, it is used both as a complex object, which contains the required behavior for bonjour, as well as a container for the settings which apply to it.

The settings of the BonjourAccount are described below:

```
BonjourAccount.id (type=SIPAddress)
```

This is not a setting, as it is the static string 'bonjour@local' which represents the id of the BonjourAccount.

```
BonjourAccount.enabled (type=bool, default=True)
```

If this setting is set to True, the account will be used. More information about this is in BonjourAccount.

```
BonjourAccount.display_name (type=str, default=None, nillable=True)
```

The contents of this setting will be sent as part of the From header when sending SIP requests.

MSRP

```
SIPSimpleSettings.msrp.transport (type=MSRPTransport, default='tcp')
```

MSRP can use either TLS or TCP and this setting controls which one should be used for the bonjour account.

RTP

```
BonjourAccount.rtp.audio_codec_list (type=AudioCodecList,
default=('speex', 'g722', 'g711', 'ilbc', 'gsm'))
```

This setting is used to specify the preferred audio codecs, which should be used for audio calls of this account. It must be a tuple containing only strings, which represent the supported codecs (speex, g722, g711, ilbc and gsm), in the order in which they are preferred.

```
BonjourAccount.rtp.srtp_encryption (type=SRTPEncryption,
default='optional')
```

The value of this setting specifies how the account requires the calls to be encrypted using SRTP. It can be one of the values 'disabled', 'optional' or 'mandatory'.

```
BonjourAccount.rtp.use_srtp_without_tls (type=bool, default=False)
```

If this setting is set to True, SRTP could be used even if the SIP signaling used to control the call is not over TLS.

TLS

```
BonjourAccount.tls.certificate (type=Path, default=None, nillable=True)
```

The path to the file that contains the certificate and its private key used to authenticate on TLS connections. It is interpreted as an absolute path, with a leading ~ expanded to the home directory of the current user. In order to access the full path to the certificate file, the normalized attribute on the setting can be used:

```
  BonjourAccount().tls.ca_list.normalized
BonjourAccount.tls.verify_server (type=bool, default=False)
```

If this setting is set to True, the middleware will verify the server's certificate when connecting via TLS.

SIPClients Settings

The SIPClients scripts use the Configuration API to extend the settings in the middleware with some application-specific settings. The following sections list these additional settings in order to provide an example for the kind of settings which, being application specific, do not find their place in the middleware and should be added by the application.

General

```
SIPSimpleSettings.user_data_directory (type=AbsolutePath,
default='~/.sipclient)
```

This is the directory, which will be used by default for storing the SIP SIMPLE data. The relative paths are calculated on runtime based on this setting, which means that if this setting is changed, all relative paths will point inside the new directory. It is a string, which must be an absolute path.

Audio

```
SIPSimpleSettings.audio.directory (type=DataPath,
default=DataPath('history'))
```

This directory will be used to store recorded audio conversations. Under this directory, a subdirectory per account with the id of the account as the name will be created. If it is set to relative path, it is taken relative to SIPSimpleSettings.user_data_directory; otherwise it is interpreted as an absolute path. In order to access the full path to the history directory, the value attribute on the setting can be used:

```
SIPSimpleSettings().audio.directory.value
```

File Transfer

```
SIPSimpleSettings.file_transfer.directory (type=DataPath,
default=DataPath('file_transfers'))
```

This directory is used to store the files obtained via MSRP file transfer. If it is set to relative path, it is taken relative to SIPSimpleSettings.user_data_directory; otherwise it is interpreted as an absolute path. In order to access the full path to the history directory, the value attribute on the setting can be used:

```
SIPSimpleSettings().file_transfer.directory.value
```

Logs

```
SIPSimpleSettings.logs.directory (type=DataPath,
default=DataPath('logs'))
```

This is the directory where the logs create by the SIP SIMPLE middleware will be stored. If it is set to relative path, it is taken relative to SIPSimpleSettings.user_data_directory; otherwise it is interpreted as an absolute path. In order to access the full path to the history directory, the value attribute on the setting can be used:

```
SIPSimpleSettings().logs.directory.value
```

SIPSimpleSettings.logs.trace_sip (type=bool, default=False)

If this setting is set to True, the SIP packets will be written to a log file named 'sip_trace.txt', inside the directory pointed by SIPSimpleSettings.logging.directory.

```
SIPSimpleSettings.logs.trace_pjsip (type=bool, default=False)
```

If this setting is set to True, the PJSIP log messages will be written to a log file named 'pjsip_trace.txt', inside the directory pointed by SIPSimpleSettings.logging.directory.

```
SIPSimpleSettings.logs.trace_msrp (type=bool, default=False)
```

If this setting is set to True, the MSRP packets will be written to a log file named 'msrp_trace.txt', inside the directory pointed by SIPSimpleSettings.logging.directory.

```
SIPSimpleSettings.logs.trace_xcap (type=bool, default=False)
```

If this setting is set to True, the XCAP packets will be written to a log file named 'xcap_trace.txt', inside the directory pointed by SIPSimpleSettings.logging.directory.

Sounds

```
SIPSimpleSettings.sounds.audio_inbound (type=AbsolutePath,
default=None, nillable=True)
```

This setting should point to a wav file, which will be played when a SIP session request is received. If it is set to None, no sound will be played.

```
SIPSimpleSettings.sounds.audio_outbound (type=AbsolutePath,
default=None, nillable=True)
```

This setting should point to a wav file, which will be used as ringtone during an outgoing SIP session request as a response to a 180 Ringing. If it is set to None, no sound will be played.

```
SIPSimpleSettings.sounds.file_sent (type=AbsolutePath, default=None,
nillable=True)
```

This setting should point to a wav file, which will be played when an outgoing file transfer is finished. If it is set to None, no sound will be played.

```
SIPSimpleSettings.sounds.file_received (type=AbsolutePath,
default=None, nillable=True)
```

This setting should point to a wav file, which will be played when an incoming file transfer is finished. If it is set to None, no sound will be played.

```
SIPSimpleSettings.sounds.message_sent (type=AbsolutePath, default=None,
nillable=True)
```

This setting is a string representing an absolute path to a wav file, which is played when a message is sent in a chat session. If it is set to None, no sound is played.

```
SIPSimpleSettings.sounds.message_received (type=AbsolutePath,
default=None, nillable=True)
```

This setting is a string representing an absolute path to a wav file, which is played when a message is received in a chat session. If it is set to None, no sound is played.

Account

Sounds

```
Account.sounds.audio_inbound (type=AbsolutePath, default=None,
nillable=True)
```

This setting should point to a wav file, which will be used to play the incoming ringtone. If it is set to None, the wav file set in SIPSimpleSettings.sounds.audio_inbound will be used instead.

BonjourAccount

Sounds

```
BonjourAccount.sounds.audio_inbound (type=AbsolutePath, default=None,
nillable=True)
```

This setting should point to a wav file which will be used as the incoming ringtone. If it is set to None, the wav file set in SIPSimpleSettings.sounds.audio_inbound will be used instead.

SIP Core API

This chapter describes the internal architecture and API of the SIP core of the sipsimple library. sipsimple is a Python package, the core of which wraps the PJSIP C library, which handles SIP signaling and audio media for the SIP SIMPLE client.

SIP stands for 'Sessions Initiation Protocol', an IETF standard described by RFC 3261. SIP is an application-layer control protocol that can establish, modify and terminate multimedia sessions such as Internet telephony calls (VoIP). Media can be added to (and removed from) an existing session.

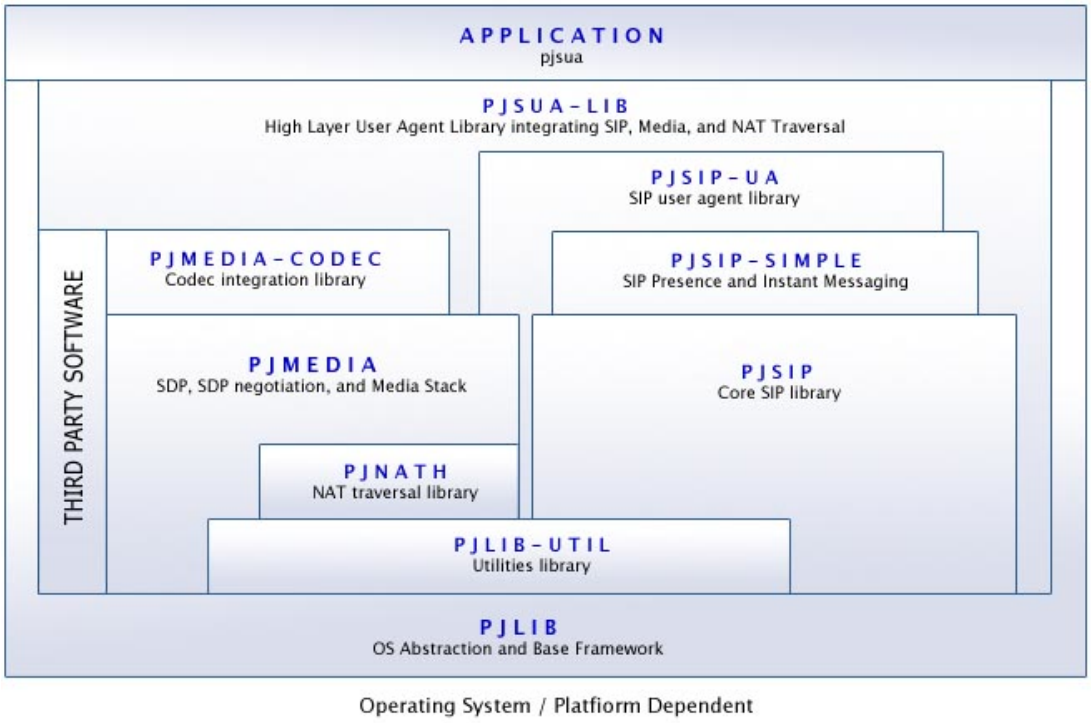
SIP transparently supports name mapping and redirection services, which supports personal mobility, users can maintain a single externally visible address identifier, which can be in the form of a standard email address or E.164 telephone number regardless of their physical network location.

SIP allows the endpoints to negotiate and combine any type of session they mutually understand like video, instant messaging (IM), file transfer, desktop sharing and provides a generic event notification system with real-time publications and subscriptions about state changes that can be used for asynchronous services like presence, message waiting indicator and busy line appearance.

For a comprehensive overview of SIP related protocols and use cases visit http://www.tech-invite.com

PJSIP library

sipsimple builds on PJSIP http://www.pjsip.org, a set of static libraries, written in C, which provide SIP signaling and media capabilities. PJSIP is considered to be the most mature and advanced open source SIP stack available. The following diagram, taken from the PJSIP documentation, illustrates the library stack of PJSIP:



The diagram shows that there is a common base library, and two more or less independent stacks of libraries, one for SIP signaling and one for SIP media. The latter also includes an abstraction layer for the sound-card. Both of these stracks are integrated in the high level library, called PJSUA.

PJSIP itself provides a high-level Python wrapper for PJSUA. Despite this, the choice was made to bypass PJSUA and write the SIP core of the sipsimple package as a Python wrapper, which directly uses the PJSIP and PJMEDIA libraries. The main reasons for this are the following:

1. PJSUA assumes a session with exactly one audio stream, whilst for the SIP SIMPLE client more advanced (i.e. low-level) manipulation of the SDP is needed.
2. What is advertised as SIMPLE functionality, it is minimal and incomplete subset of it. Only page mode messaging using SIP MESSAGE method and basic device status presence are possible, while session mode IM and rich presence are desired.
3. PJSUA integrates the decoding and encoding of payloads (e.g. presence related XML documents), while in the SIP SIMPLE client this should be done at a high level, not by the SIP stack.

PJSIP itself is by nature asynchronous. In the case of PJSIP it means that in general there will be one thread which handles reception and transmission of SIP signaling

messages by means of a polling function which is continually called by the application. Whenever the application performs some action through a function, this function will return immediately. If PJSIP has a result for this action, it will notify the application by means of a callback function in the context of the polling function thread.

NOTE

> Currently the core starts the media handling as a separate C thread to avoid lag caused by the GIL. The sound-card also has its own C thread.

Architecture

The sipsimple core wrapper itself is mostly written using Cython (formerly Pyrex). It allows a Python-like file with some added C manipulation statements to be compiled to C. This in turn compiles to a Python C extension module, which links with the PJSIP static libraries.

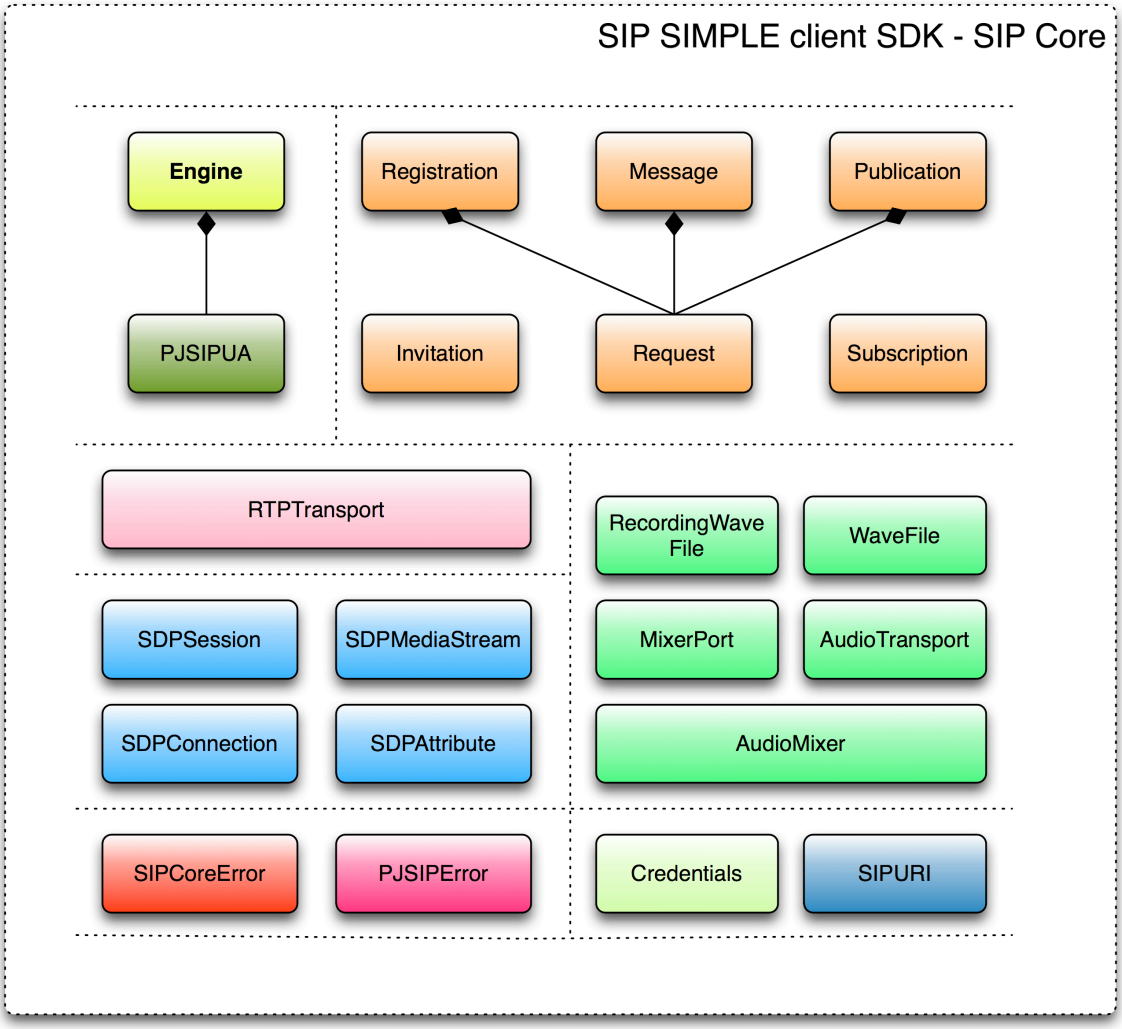
The SIP core part of the sipsimple Python library resides in the sipsimple.core package. This package aggregates three modules, sipsimple.core._core, sipsimple.core._engine and sipsimple.core._primitives. The former is a Python C extension module which makes wrappers around PJSIP objects available in Python, while the latter two contain SIP core objects written in Python. All core objects should be accessed from the enclosing sipsimple.core module. The following list enumerates the various SIP core objects available:

1. The Engine class which is a Python wrapper around the low-level PJSIPUA class. The latter represents the SIP endpoint and manages the initialization and destruction of all the PJSIP libraries. It is also the central management point to the SIP core. The application should not use the PJSIPUA class directly, but rather through the wrapping Engine, which is a singleton class.
2. Utility classes used throughout the core:
    - frozenlist and frozendict: classes which relate respectively to list and dict similarly to how the standard frozenset relates to set.
3. Helper classes which represent a structured collection of data which is used throughout the core:
    - BaseSIPURI, SIPURI and FrozenSIPURI
    - BaseCredentials, Credentials and FrozenCredentials
4. SDP manipulation classes, which directly wrap the PJSIP structures representing either the parsed or to be generated SDP:
    - BaseSDPSession, SDPSession and FrozenSDPSession
    - BaseSDPMediaStream, SDPMediaStream and FrozenSDPMediaStream
    - BaseSDPConnection, SDPConnection and FrozenSDPConnection
    - SDPAttributeList and FrozenSDPAttributeList
    - BaseSDPAttribute, SDPAttribute and FrozenSDPAttribute
5. Audio handling classes:
    - AudioMixer
    - MixerPort
    - WaveFile
    - RecordingWaveFile
    - ToneGenerator
6. Media transport handling classes, using the functionality built into PJMEDIA:
    - RTPTransport
    - AudioTransport
7. SIP signalling related classes:
    - Request and IncomingRequest: low-level transaction support
    - Invitation: INVITE-dialog support
    - Subscription and IncomingSubscription: SUBSCRIBE-dialog support (including NOTIFY handling within the SUBSCRIBE dialog)
    - Referral and IncomingReferral: REFER-dialog support (including NOTIFY handling within the dialog)
    -
    - Registration: Python object based on Request for REGISTER support

- Message: Python object based on Request for MESSAGE support
- Publication: Python object based on Request for PUBLISH support

8. Exceptions:
   - SIPCoreError: generic error used throught the core
   - PJSIPError: subclass of SIPCoreError which offers more information related to errors from PJSIP
   - PJSIPTLSError: subclass of PJSIPError to distinguish between TLS-related errors and the rest
   - SIPCoreInvalidStateError: subclass of SIPCoreError used by objects which are based on a state-machine

Most of the objects cannot be used until the Engine has been started. The following diagram illustrates these classes:



Most of the SIP core does not allow duck-typing due to the nature of the integration between it and PJSIP. If these checks had not been employed, any errors could have resulted in a segmentation fault and a core dump. This also explains why several objects have a Frozen counterpart: the frozen objects are simply immutable versions of their non-frozen variants which make sure that low-level data is kept consistent and cannot be modified from Python. The Base versions are just base classes for the

frozen and non-frozen versions provided mostly for convinience: they cannot be instantiated.

Integration

The core itself has one Python dependency, the application module, which in turn depends on the zope.interface module. These modules should be present on the system before the core can be used. An application that uses the SIP core must use the notification system provided by the application module in order to receive notifications from it. It does this by creating one or more classes that act as an observer for particular messages and registering it with the NotificationCenter, which is a singleton class. This means that any call to instance an object from this class will result in the same object. As an example, this bit of code will create an observer for logging messages only:

```
from zope.interface import implements
from application.notification import NotificationCenter, IObserver

class SIPEngineLogObserver(object):
    implements(IObserver)

    def handle_notification(self, notification):
        print "%(timestamp)s (%(level)d) %(sender)14s: %(message)s" %
notification.data.__dict__

log_observer = SIPEngineLogObserver()
notification_center = NotificationCenter()
notification_center.add_observer(log_observer, name="SIPEngineLog")
```

Each notification object has three attributes:

sender

The object that sent the notification. For generic notifications the sender will be the Engine instance, otherwise the relevant object.

name

The name describing the notification. Most of the notifications in the core have the prefix "SIP".

data

An instance of application.notification.NotificationData or a subclass of it. The attributes of this object provide additional data about the notification. Notifications described in this document will also have the data attributes described.

Besides setting up the notification observers, the application should import the relevant objects from the sipsimple.core module. It can then instantiate the Engine class, which is also a singleton, and start the PJSIP worker thread by calling Engine.start(), optionally providing a number of initialization options. Most of these options can later be changed at runtime, by setting attributes of the same name on the Engine object. The application may then instantiate one of the SIP primitive classes and perform operations on it.

When starting the Engine class, the application can pass a number of keyword arguments that influence the behaviour of the SIP endpoint. For example, the SIP network ports may be set through the local_udp_port, local_tcp_port and local_tls_port arguments. The UDP/RTP ports are described by a range of ports through rtp_port_range, two of which will be randomly selected for each RTPTransport object and effectively each audio stream.

The methods called on the SIP primitive objects and the Engine object (proxied to the PJSIPUA instance) may be called from any thread. They will return immediately and any delayed result, such as results depending on network traffic, will be returned later using a notification. In this manner the SIP core continues the asynchronous pattern of PJSIP. If there is an error in processing the request, an instance of SIPCoreError, or its subclass PJSIPError will be raised. The former will be raised whenever an error occurs inside the core, the latter whenever an underlying PJSIP function returns an error. The PJSIPError object also contains a status attribute, which is the PJSIP errno as an integer.

As a very basic example, one can REGISTER for a sip account by typing the following lines on a Python console:

```
from sipsimple.core import ContactHeader, Credentials, Engine,
Registration, RouteHeader, SIPURI
e = Engine()
e.start()
identity = FromHeader(SIPURI(user="alice", host="example.com"),
display_name="Alice")
cred = Credentials("alice", "mypassword")
reg = Registration(identity, credentials=cred)
reg.register(ContactHeader(SIPURI("127.0.0.1",port=12345)),
RouteHeader(SIPURI("1.2.3.4", port=5060)))
```

Note that in this example no observer for notifications from this Registration object are registered, so the result of the operation cannot be seen. Also note that this will not keep the registration registered when it is about to expire, as it is the application's responsibility. See the Registration documentation for more details.

Another convention that is worth mentioning at this point is that the SIP core will never perform DNS lookups. For the sake of flexibility, it is the responsibility of the application to do this and pass the result to SIP core objects using the RouteHeader object, indicating the destination IP address, port and transport the resulting SIP request should be sent to. The {{{sipsimple.lookup}}} module of the middleware can be used to perform DNS lookups according to RFC3263.

Components

Engine

As explained above, this singleton class needs to be instantiated by the application using the SIP core of sipsimple and represents the whole SIP core stack. Once the start() method is called, it instantiates the core.PJSIPUA object and will proxy attribute and methods from it to the application.

attributes

```
default_start_options (class attribute)
```

This dictionary is a class attribute that describes the default values for the initialization options passed as keyword arguments to the start() method. Consult this method for documentation of the contents.

```
is_running
```

A boolean property indicating if the Engine is running and if it is safe to try calling any proxied methods or attributes on it.

methods

```
__init__(self)
```

This will either create the Engine if it is called for the first time or return the one Engine instance if it is called subsequently.

```
start(self, **kwargs)
```

Initialize all PJSIP libraries based on the keyword parameters provided and start the PJSIP worker thread. If this fails an appropriate exception is raised. After the Engine has been started successfully, it can never be started again after being stopped. The keyword arguments will be discussed here. Many of these values are also readable as (proxied) attributes on the Engine once the start() method has been called. Many of them can also be set at runtime, either by modifying the attribute or by calling a particular method. This will also be documented for each argument in the following list of options.

udp_port: (Default: 0)

The local UDP port to listen on for UDP datagrams. If this is 0, a random port will be chosen. If it is None, the UDP transport is disabled, both for incoming and outgoing traffic. As an attribute, this value is read-only, but it can be changed at runtime using the set_udp_port() method.

tcp_port: (Default: 0)

The local TCP port to listen on for new TCP connections. If this is 0, a random port will be chosen. If it is None, the TCP transport is disabled, both for incoming and outgoing traffic. As an attribute, this value is read-only, but it can be changed at runtime using the set_tcp_port() method.

tls_port: (Default: 0)

The local TCP port to listen on for new TLS over TCP connections. If this is 0, a random port will be chosen. If it is None, the TLS transport is disabled, both for incoming and outgoing traffic. As an attribute, this value is read-only, but it can be changed at runtime using the set_tls_options() method, as internally the TLS transport needs to be restarted for this operation.

tls_protocol: (Default: "TLSv1")

This string describes the (minimum) TLS protocol that should be used. Its values should be either None, "SSLv2", "SSLv23", "SSLv3" or "TLSv1". If None is specified, the PJSIP default will be used, which is currently "TLSv1".

tls_verify_server: (Default: False)

This boolean indicates whether PJSIP should verify the certificate of the server against the local CA list when making an outgoing TLS connection. As an attribute, this value is read-only, but it can be changed at runtime using the set_tls_options() method, as internally the TLS transport needs to be restarted for this operation.

tls_ca_file: (Default: None)

This string indicates the location of the file containing the local list of CA certificates, to be used for TLS connections. If this is set to None, no CA certificates will be read. As an attribute, this value is read-only, but it can be changed at runtime using the set_tls_options() method, as internally the TLS transport needs to be restarted for this operation.

tls_cert_file: (Default: None)

This string indicates the location of a file containing the TLS certificate to be used for TLS connections. If this is set to None, no certificate file will be read. As an attribute, this value is read-only, but it can be changed at runtime using the set_tls_options() method, as internally the TLS transport needs to be restarted for this operation.

tls_privkey_file: (Default: None)

This string indicates the location of a file containing the TLS private key associated with the above mentioned certificated to be used for TLS connections. If this is set to None, no private key file will be read. As an attribute, this value is read-only, but it can be changed at runtime using the set_tls_options() method, as internally the TLS transport needs to be restarted for this operation.

tls_timeout: (Default: 1000)

The timeout value for a TLS negotiation in milliseconds. Note that this value should be reasonably small, as a TLS negotiation blocks the whole PJSIP polling thread. As an attribute, this value is read-only, but it can be changed at runtime using the set_tls_options() method, as internally the TLS transport needs to be restarted for this operation.

user_agent: (Default: "sipsimple-%version-pjsip-%pjsip_version-r%pjsip_svn_revision")

This value indicates what should be set in the User-Agent header, which is included in each request or response sent. It can be read and set directly as an attribute at runtime.

log_level: (Default: 5)

This integer dictates the maximum log level that may be reported to the application by PJSIP through the SIPEngineLog notification. By default the maximum amount of logging information is reported. This value can be read and set directly as an attribute at runtime.

trace_sip: (Default: False)

This boolean indicates if the SIP core should send the application SIP messages as seen on the wire through the SIPEngineSIPTrace notification. It can be read and set directly as an attribute at runtime.

rtp_port_range: (Default: (40000, 40100))

This tuple of two integers indicates the range to select UDP ports from when creating a new RTPTransport object, which is used to transport media. It can be read and set directly as an attribute at runtime, but the ports of previously created RTPTransport objects remain unaffected.

codecs: (Default: ["speex", "G722", "PCMU", "PCMA", "iLBC", "GSM"])

This list specifies the codecs to use for audio sessions and their preferred order. It can be read and set directly as an attribute at runtime. Note that this global option can be overridden by an argument passed to AudioTransport.__init__(). The strings in this list is case insensitive.

events: (Default: <some sensible events>)

PJSIP needs a mapping between SIP SIMPLE event packages and content types. This dictionary provides some default packages and their event types. As an attribute, this value is read-only, but it can be changed at runtime using the add_event() method.

incoming_events: (Default: set())

A list that specifies for which SIP SIMPLE event packages the application wishes to receive IncomingSubscribe objects. When a SUBSCRIBE request is received containing an event name that is not in this list, a 489 "Bad event" response is internally generated. When the event is in the list, an IncomingSubscribe object is created based on the request and passed to the application by means of a notification. Note that each of the events specified here should also be a key in the events dictionary argument. As an attribute, this value is read-only, but it can be changed at runtime using the add_incoming_event() and remove_incoming_event() methods.

incoming_requests: (Default: set())

A set of methods for which IncomingRequest objects are created and sent to the application if they are received. Note that receiving requests using the INVITE, SUBSCRIBE, ACK or BYE methods in this way is not allowed. Requests using the OPTIONS or MESSAGE method are handled internally, but may be overridden.

```
stop(self)
```

Stop the PJSIP worker thread and unload all PJSIP libraries. Note that after this all references to SIP core objects can no longer be used, these should be properly removed by the application itself before stopping the Engine. Also note that, once stopped the Engine cannot be started again. This method is automatically called when the Python interpreter exits.

proxied attributes

Besides all the proxied attributes described for the __init__ method above, thse other attributes are provided once the Engine has been started.

```
input_devices
```

This read-only attribute is a list of strings, representing all audio input devices on the system that can be used. One of these device names can be passed as the input_device argument when creating a AudioMixer object.

```
output_devices
```

This read-only attribute is a list of strings, representing all audio output devices on the system that can be used. One of these device names can be passed as the output_device argument when creating a AudioMixer object.

```
sound_devices
```

This read-only attribute is a list of strings, representing all audio sound devices on the system that can be used.

```
available_codecs
```

A read-only list of codecs available in the core, regardless of the codecs configured through the codecs attribute.

proxied methods

```
add_event(self, event, accept_types)
```

Couple a certain event package to a list of content types. Once added it cannot be removed or modified.

```
add_incoming_event(self, event)
```

Adds a SIP SIMPLE event package to the set of events for which the Engine should create an IncomingSubscribe object when a SUBSCRIBE request is received. Note that this event should be known to the Engine by means of the events attribute.

```
remove_incoming_event(self, event)
```

Removes an event from the incoming_events attribute. Incoming SUBSCRIBE requests with this event package will automatically be replied to with a 489 "Bad Event" response.

```
add_incoming_request(self, method)
```

Add a method to the set of methods for which incoming requests should be turned into IncomingRequest objects. For the rules on which methods are allowed, see the description of the Engine attribute above.

```
remove_incoming_request(self, method)
```

Removes a method from the set of methods that should be received.

```
detect_nat_type(self, stun_server_address, stun_server_port=3478,
user_data=None)
```

Will start a series of STUN requests which detect the type of NAT this host is behind. The stun_server_address parameter indicates the IP address or hostname of the STUN server to be used and stun_server_port specifies the remote UDP port to use. When the type of NAT is detected, this will be reported back to the application by means of a SIPEngineDetectedNATType notification, including the user_data object passed with this method.

```
set_udp_port(self, value)
```

Update the local_udp_port attribute to the newly specified value.

```
set_tcp_port(self, value)
```

Update the local_tcp_port attribute to the newly specified value.

```
set_tls_options(self, local_port=None, protocol="TLSv1",
verify_server=False, ca_file=None, cert_file=None,
privkey_file=None, timeout=1000)
```

Calling this method will (re)start the TLS transport with the specified arguments, or stop it in the case that the local_port argument is set to None. The semantics of the arguments are the same as on the start() method.

notifications

Notifications sent by the Engine are notifications that are related to the Engine itself or unrelated to any SIP primitive object. They are described here including the data attributes that is included with them.

```
SIPEngineWillStart
```

This notification is sent when the Engine is about to start.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPEngineDidStart
```

This notification is sent when the Engine is has just been started.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPEngineDidFail
```

This notification is sent whenever the Engine has failed fatally and either cannot start or is about to stop. It is not recommended to call any methods on the Engine at this point.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPEngineWillEnd
```

This notification is sent when the Engine is about to stop because the application called the stop() method. Methods on the Engine can be called at this point, but anything that has a delayed result will probably not return any notification.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPEngineDidEnd
```

This notification is sent when the Engine was running and is now stopped, either because of failure or because the application requested it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPEngineLog
```

This notification is a wrapper for PJSIP logging messages. It can be used by the application to output PJSIP logging to somewhere meaningful, possibly doing filtering based on log level.

timestamp:

A datetime.datetime object representing the time when the log message was output by PJSIP.

sender:

The PJSIP module that originated this log message.

level:

The logging level of the message as an integer. Currently this is 1 through 5, 1 being the most critical.

message:

The actual log message.

```
SIPEngineSIPTrace
```

Will be sent only when the do_siptrace attribute of the Engine instance is set to True. The notification data attributes will contain the SIP messages as they are sent and received on the wire.

timestamp:

A datetime.datetime object indicating when the notification was sent.

received:

A boolean indicating if this message was sent from or received by PJSIP (i.e. the direction of the message).

source_ip:

The source IP address as a string.

source_port:

The source port of the message as an integer.

destination_ip:

The destination IP address as a string.

source_port:

The source port of the message as an integer.

data:

The contents of the message as a string.

For received message the destination_ip and for sent messages the source_ip may not be reliable.

```
SIPEngineDetectedNATType
```

This notification is sent some time after the application request the NAT type this host behind to be detected using a STUN server. Note that there is no way to associate a request to do this with a notification, although every call to the detect_nat_type() method will generate exactly one notification.

timestamp:

A datetime.datetime object indicating when the notification was sent.

succeeded:

A boolean indicating if the NAT detection succeeded.

user_data:

The user_data argument passed while calling the detect_nat_type() method. This can be any object and could be used for matching requests to responses.

nat_type:

A string describing the type of NAT found. This value is only present if NAT detection succeeded.

error:

A string indicating the error that occurred while attempting to detect the type of NAT. This value only present if NAT detection did not succeed.

```
SIPEngineGotException
```

This notification is sent whenever there is an unexpected exception within the PJSIP working thread. The application should show the traceback to the user somehow. An exception need not be fatal, but if it is it will be followed by a SIPEngineDidFail notification.

timestamp:

A datetime.datetime object indicating when the notification was sent.

traceback:

A string containing the traceback of the exception. In general this should be printed on the console.

```
SIPEngineGotMessage
```

This notification is sent whenever the Engine receives a MESSAGE request.

request_uri:

The request URI of the MESSAGE request as a SIPURI object.

from_header:

The From header of the MESSAGE request as a FrozenFromHeader object.

to_header:

The To header of the MESSAGE request as a FrozenToHeader object.

content_type:

The Content-Type header value of the MESSAGE request as a ContentType object.

headers:

The headers of the MESSAGE request as a dict. Each SIP header is represented in its parsed for as long as PJSIP supports it. The format of the parsed value depends on the header.

body:

The body of the MESSAGE request as a string, or None if no body was included.

timestamp:

A datetime.datetime object indicating when the notification was sent.

SIPURI

These are helper objects for representing a SIP URI. This object needs to be used whenever a SIP URI should be specified to the SIP core. It supports comparison to other SIPURI objects using the == and != expressions. As all of its attributes are set by the __init__ method, the individual attributes will not be documented here. The FrozenSIPURI object does not allow any of its attributes to be changed after initialization.

methods

```
__init__(self, host, user=None, port=None, display=None,
secure=False, parameters=None, headers=None)
```

Creates the SIPURI object with the specified parameters as attributes. host is the only mandatory attribute.

host:

The host part of the SIP URI as a string.

user:

The username part of the SIP URI as a string, or None if not set.

port:

The port part of the SIP URI as an int, or None or 0 if not set.

display:

The optional display name of the SIP URI as a string, or None if not set.

secure:

A boolean indicating whether this is a SIP or SIPS URI, the latter being indicated by a value of True.

parameters:

The URI parameters. represented by a dictionary.

headers:

The URI headers, represented by a dictionary.

```
__str__(self)
```

The special Python method to represent this object as a string, the output is the properly formatted SIP URI.

```
new(cls, sipuri)
```

Classmethod that returns an instance of the class on which it has been called which is a copy of the sipuri object (which must be either a SIPURI or a FrozenSIPURI).

```
parse(cls, uri_str)
```

Classmethod that returns an instance of the class on which it has been called which is represents the parsed version of the URI provided as a string. A SIPCoreError is raised if the string is invalid or if the Engine has not been started yet.

```
matches(self, address)
```

This method returns True or False depending on whether the string address contains a SIP address whose components are a subset of the components of self.

For example:

```
SIPURI.parse('sip:alice@example.org:54321;transport=tls').matches('alic
e@example.org')
```

returns True while

```
SIPURI.parse('sip:alice@example.org;transport=tls').matches('sips:alice
@example.org')
```

returns False.

Credentials

The Credentials and FrozenCredentails simple objects represent authentication credentials for a particular SIP account. These can be included whenever creating a SIP primitive object that originates SIP requests. The attributes of this object are the same as the arguments to the __init__ method. Note that the domain name of the SIP account is not stored on this object.

methods

```
__init__(self, username, password)
```

Creates the Credentials object with the specified parameters as attributes. Each of these attributes can be accessed and changed on the object once instanced.

username:

A string representing the username of the account for which these are the credentials.

password:

The password for this SIP account as a string.

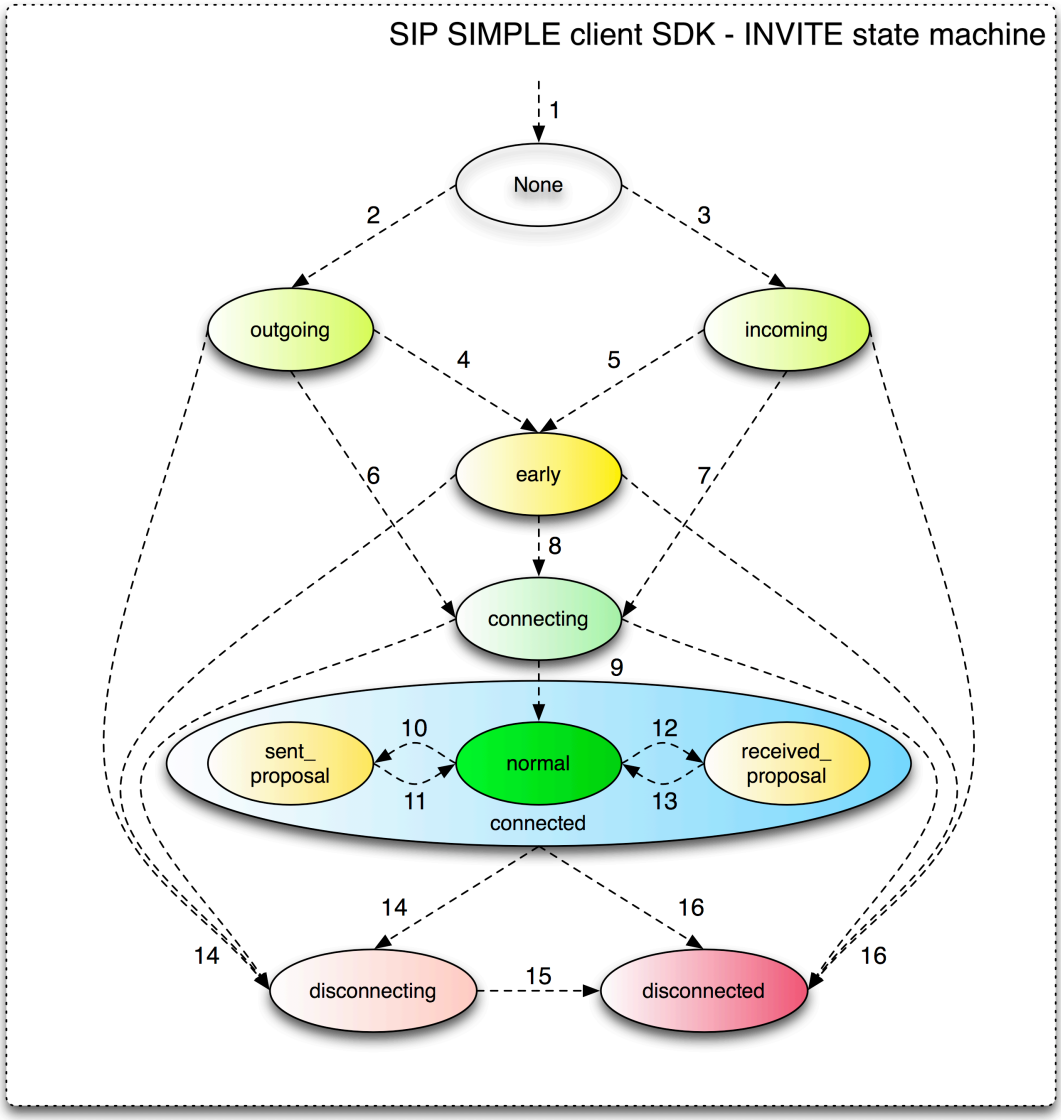
```
new(cls, credentials)
```

Classmethod that returns an instance of the class on which it has been called which is a copy of the credentials object (which must be either a Credentials or a FrozenCredentials).

Invitation

The Invitation class represents an INVITE session, which governs a complete session of media exchange between two SIP endpoints from start to finish. It is implemented to be agnostic to the media stream or streams negotiated, which is achieved by using the SDPSession class and its companion classes, which directly represents the parsed SDP. The Invitation class represents both incoming and outgoing sessions.

The state machine contained in each Invitation object is based on the one used by the underlying PJSIP pjsip_inv_session object. In order to represent re-INVITEs and user-requested disconnections, three more states have been added to this state machine. The progression through this state machine is fairly linear and is dependent on whether this is an incoming or an outgoing session. State changes are triggered either by incoming or by outgoing SIP requests and responses. The states and the transitions between them are shown in the following diagram:



The state changes of this machine are triggered by the following:

1. An Invitation object is newly created, either by the application for an outgoing session, or by the core for an incoming session.
2. The application requested an outgoing session by calling the send_invite() method and and initial INVITE request is sent.
3. A new incoming session is received by the core. The application should look out for state change to this state in order to be notified of new incoming sessions.
4. A provisional response (1xx) is received from the remove party.
5. A provisional response (1xx) is sent to the remote party, after the application called the respond_to_invite_provisionally() method.
6. A positive final response (2xx) is received from the remote party.
7. A positive final response (2xx) is sent to the remote party, after the application called the accept_invite() method.
8. A positive final response (2xx) is sent or received, depending on the orientation of the session.
9. An ACK is sent or received, depending on the orientation of the session. If the ACK is sent from the local to the remote party, it is initiated by PJSIP, not by a call from the application.
10. The local party sent a re-INVITE to the remote party by calling the send_reinvite() method.
11. The remote party has sent a final response to the re-INVITE.
12. The remote party has sent a re-INVITE.
13. The local party has responded to the re-INVITE by calling the respond_to_reinvite() method.
14. The application requests that the session ends by calling the end() method.
15. A response is received from the remote party to whichever message was sent by the local party to end the session.
16. A message is received from the remote party which ends the session.

The application is notified of a state change in either state machine through the SIPInvitationChangedState notification, which has as data the current and previous states. If the event is triggered by and incoming message, extensive data about that message, such as method/code, headers and body, is also included with the notification. The application should compare the previous and current states and perform the appropriate action.

An Invitiation object also emits the SIPInvitationGotSDPUpdate notification, which indicates that SDP negotiation between the two parties has been completed. This will occur (at least) once during the initial session negotiation steps, during re-INVITEs in both directions and whenever an UPDATE request is received. In the last case, the Invitation object will automatically include the current local SDP in the response.

attributes

```
state
```

The state the Invitation state machine is currently in. See the diagram above for possible states. This attribute is read-only.

```
sub_state
```

The sub-state the Invitation state machine is currently in. See the diagram above for possible states. This attribute is read-only.

```
directing
```

A string with the values "incoming" or "outgoing" depending on the direction of the original INVITE request.

```
credentials
```

The SIP credentials needed to authenticate at the SIP proxy in the form of a FrozenCredentials object. If this Invitation object represents an incoming INVITE session this attribute will be None. On an outgoing session this attribute will be None if it was not specified when the object was created. This attribute is set on object instantiation and is read-only.

```
from_header
```

The From header of the caller represented by a FrozenFromHeader object. If this is an outgoing INVITE session, this is the from_header from the send_invite() method. Otherwise the URI is taken from the From: header of the initial INVITE. This attribute is set on object instantiation and is read-only.

```
to_header
```

The To header of the callee represented by a FrozenToHeader object. If this is an outgoing INVITE session, this is the to_header from the send_invite() method. Otherwise the URI is taken from the To: header of the initial INVITE. This attribute is set on object instantiation and is read-only.

```
local_identity
```

The From or To header representing the local identity used in this session. If the original INVITE was incoming, this is the same as to_header, otherwise it will be the same as from_header.

```
remote_identity
```

The From or To header representing the remote party in this session. If the original INVITE was incoming, this is the same as from_header, otherwise it will be the same as to_header.

```
route_header
```

The outbound proxy that was requested to be used in the form of a FrozenRouteHeader object, including the desired transport. If this Invitation object

represents an incoming INVITE session this attribute will always be None. This attribute is set on object instantiation and is read-only.

```
call_id
```

The call ID of the INVITE session as a read-only string. In the NULL and DISCONNECTED states, this attribute is None.

```
transport
```

A string indicating the transport used for the application. This can be "udp", "tcp" or "tls".

```
local_contact_header
```

The Contact header that the local side provided to the remote side within this INVITE session as a FrozenContactHeader object. Note that this can either be set on object creation or updated using the send_reinvite() method.

```
remote_contact_header
```

The Contact header that the remote side provided to us within this INVITE session as a FrozenContactHeader object.

```
call_id
```

A string representing the Call-Id header value of this INVITE dialog.

```
remote_user_agent
```

A string representing the remote user agent taken from the User-Agent or Server headers (depending on the direction of the original INVITE).

```
sdp.proposed_local
```

The currently proposed sdp by the local party in the form of a FrozenSDPSession object. This attribute is None when an SDP proposal is not in progress.

```
sdp.proposed_remote
```

The currently proposed sdp by the remote party in the form of a FrozenSDPSession object. This attribute is None when an SDP proposal is not in progress.

```
sdp.active_local
```

The currently active sdp of the local party in the form of a FrozenSDPSession object. This attribute is None if no SDP proposal has succeeded before.

```
sdp.active_remote
```

The currently active sdp of the remote party in the form of a FrozenSDPSession object. This attribute is None if no SDP proposal has succeeded before.

```
peer_address
```

This read-only attribute contains the remote endpoint IP and port information. It can be accessed by accessing this object's ip and port attributes.

methods

```
__init__(self)
```

Creates a new Invitation object.

```
send_invite(self, request_uri, from_header, to_header,
route_header, contact_header, sdp, credentials=None,
extra_headers=[], timeout=None)
```

request_uri:

Request URI to be set inthe outgoing INVITE request.

from_header:

The identity of the local account in the form of a FromHeader object.

to_header:

The identity we want to send the INVITE to, represented as a ToHeader object.

route_header:

The outbound proxy to use in the form of a RouteHeader object. This includes the desired transport to use.

contact_header:

The Contact header to include in the INVITE request, a ContactHeader object.

sdp:

The SDP to send as offer to the remote party.

credentials:

The optional SIP credentials needed to authenticate at the SIP proxy in the form of a Credentials object.

extra_headers:

Any extra headers that should be included in the INVITE request in the form of a list of header objects.

timeout:

How many seconds to wait for the remote party to reply before changing the state to DISCONNECTED and internally replying with a 408, as an int or a float.

If this is set to None, the default PJSIP timeout will be used, which appears to be slightly longer than 30 seconds.

```
send_response(self, code, reason, contact_header, sdp,
extra_headers)
```

Send a response to a INVITE request.

code:

The code of the response to use as an int.

reason:

The reason of the response as a str.

contact_header:

The Contact header to include in the response, a ContactHeader object.

sdp:

The SDP to send as offer/response to the remote party.

extra_headers:

Any extra headers that should be included in the response in the form of a list of header objects.

```
send_reinvite(self, contact_header=None, sdp=None,
extra_header=[])
```

contact_header:

The Contact header if it needs to be changed by the re-INVITE or None if it shouldn't be changed; a BaseContactHeader object.

sdp:

The SDP to send as offer to the remote party or None if the re-INVITE should not change the SDP; a BaseSDPSession object.

extra_headers:

Any extra headers that should be included in the response in the form of a list of header objects.

```
cancel_reinvite(self)
```

Send a CANCEL after a re-INVITE has been sent to cancel the action of the re-INVITE.

```
end(self, extra_headers=[], timeout=None)
```

This moves the INVITE state machine into the DISCONNECTING state by sending the necessary SIP request. When a response from the remote party is received, the state

machine will go into the DISCONNECTED state. Depending on the current state, this could be a CANCEL or a BYE request.

extra_headers:

Any extra headers that should be included in the request or response in the form of a dict.

timeout:

How many seconds to wait for the remote party to reply before changing the state to DISCONNECTED, as an int or a float. If this is set to None, the default PJSIP timeout will be used, which currently appears to be 3.5 seconds for an established session.

notifications

```
SIPInvitationChangedState
```

This notification is sent by an Invitation object whenever its state machine changes state.

timestamp:

A datetime.datetime object indicating when the notification was sent.

prev_state:

The previous state of the INVITE state machine.

prev_sub_state:

The previous sub-state of the INVITE state machine.

state:

The new state of the INVITE state machine, which may be the same as the previous state.

sub_state:

The new sub-state of teh INVITE state machine, which may be the same as the previous sub-state.

method: (only if the state change got triggered by an incoming SIP request)

The method of the SIP request as a string.

request_uri: (only if the state change got triggered by an incoming SIP request)

The request URI of the SIP request as a SIPURI object.

code: (only if the state change got triggered by an incoming SIP response or internal timeout or error)

The code of the SIP response or error as an int.

reason: (only if the state change got triggered by an incoming SIP response or internal timeout or error)

The reason text of the SIP response or error as a string.

headers: (only if the state change got triggered by an incoming SIP request

or response)

The headers of the SIP request or response as a dict. Each SIP header is represented in its parsed for as long as PJSIP supports it. The format of the parsed value depends on the header.

body: (only if the state change got triggered by an incoming SIP request or response)

The body of the SIP request or response as a string, or None if no body was included. The content type of the body can be learned from the Content-Type: header in the headers argument.

```
SIPInvitationGotSDPUpdate
```

This notification is sent by an Invitation object whenever SDP negotiation has been performed. It should be used by the application as an indication to start, change or stop any associated media streams.

timestamp:

A datetime.datetime object indicating when the notification was sent.

succeeded:

A boolean indicating if the SDP negotiation has succeeded.

error: (only if SDP negotiation did not succeed)

A string indicating why SDP negotiation failed.

local_sdp: (only if SDP negotiation succeeded)

A SDPSession object indicating the local SDP that was negotiated.

remote_sdp: (only if SDP negotiation succeeded)

A SDPSession object indicating the remote SDP that was negotiated.

SDPSession

SDP stands for Session Description Protocol. Session Description Protocol (SDP) is a format for describing streaming media initialization parameters in an ASCII string. SDP is intended for describing multimedia communication sessions for the purposes of session announcement, session invitation, and other forms of multimedia session initiation. It is an IETF standard described by RFC 4566. RFC 3264 defines an Offer/Answer Model with the Session Description Protocol (SDP), 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.

SDPSession and FrozenSDPSession objects directly represent the contents of a SDP body, as carried e.g. in an INVITE request, and is a simple wrapper for the PJSIP pjmedia_sdp_session structure. They can be passed to those methods of an Invitation object that result in transmission of a message that includes SDP, or are passed to the application through a notification that is triggered by reception of a message that includes SDP. A (Frozen)SDPSession object may contain (Frozen)SDPMediaStream, (Frozen)SDPConnection and (Frozen)SDPAttribute objects. It supports comparison to other (Frozen)SDPSession objects using the == and != expressions. As all the attributes of the (Frozen)SDPSession class are set by attributes of the __init__ method, they will be documented along with that method.

methods

```
__init__(self, address, id=None, version=None, user="-",
net_type="IN", address_type="IP4", name=" ", info=None,
connection=None, start_time=0, stop_time=0, attributes=None,
media=None)
```

Creates the SDPSession object with the specified parameters as attributes. Each of these attributes can be accessed and changed on the object once instanced.

address:

The address that is contained in the "o" (origin) line of the SDP as a string.

id:

The session identifier contained in the "o" (origin) line of the SDP as an int. If this is set to None on init, a session identifier will be generated.

version:

The version identifier contained in the "o" (origin) line of the SDP as an int. If this is set to None on init, a version identifier will be generated.

user:

The user name contained in the "o" (origin) line of the SDP as a string.

net_type:

The network type contained in the "o" (origin) line of the SDP as a string.

address_type:

The address type contained in the "o" (origin) line of the SDP as a string.

name:

The contents of the "s" (session name) line of the SDP as a string.

info:

The contents of the session level "i" (information) line of the SDP as a string. If this is None or an empty string, the SDP has no "i" line.

connection:

The contents of the "c" (connection) line of the SDP as a (Frozen)SDPConnection object. If this is set to None, the SDP has no session level "c" line.

start_time:

The first value of the "t" (time) line of the SDP as an int.

stop_time:

The second value of the "t" (time) line of the SDP as an int.

attributes:

The session level "a" lines (attributes) in the SDP represented by a list of (Frozen)SDPAttribute objects.

media:

The media sections of the SDP represented by a list of (Frozen)SDPMediaStream objects.

```
new(cls, sdp_session)
```

Classmethod that returns an instance of the class on which it has been called which is a copy of the sdp_session object (which must be either a SDPSession or a FrozenSDPSession).

attributes

```
has_ice_proposal
```

This read-only attribute returns True if the SDP contains any attributes which indicate the existence of an ice proposal and False otherwise.

SDPMediaStream

The SDPMediaStream and FrozenSDPMediaStream objects represent the contents of a media section of a SDP body, i.e. a "m" line and everything under it until the next "m" line. It is a simple wrapper for the PJSIP pjmedia_sdp_media structure. One or more (Frozen)SDPMediaStream objects are usually contained in a (Frozen)SDPSession object. It supports comparison to other (Frozen)SDPMedia objects using the == and != expressions. As all the attributes of this class are set by attributes of the __init__ method, they will be documented along with that method.

methods

```
__init__(self, media, port, transport, port_count=1,
formats=None, info=None, connection=None, attributes=None)
```

Creates the SDPMedia object with the specified parameters as attributes. Each of these attributes can be accessed and changed on the object once instanced.

media:

The media type contained in the "m" (media) line as a string.

port:

The transport port contained in the "m" (media) line as an int.

transport:

The transport protocol in the "m" (media) line as a string.

port_count:

The port count in the "m" (media) line as an int. If this is set to 1, it is not included in the SDP.

formats:

The media formats in the "m" (media) line represented by a list of strings.

info:

The contents of the "i" (information) line of this media section as a string. If this is None or an empty string, the media section has no "i" line.

connection:

The contents of the "c" (connection) line that is somewhere below the "m" line of this section as a (Frozen)SDPConnection object. If this is set to None, this media section has no "c" line.

attributes:

The "a" lines (attributes) that are somewhere below the "m" line of this section represented by a list of (Frozen)SDPAttribute objects.

```
new(cls, sdp_media)
```

Classmethod that returns an instance of the class on which it has been called which is a copy of the sdp_media object (which must be either a SDPMediaStream or a FrozenSDPMediaStream).

attributes

`direction`

This is a convenience read-only attribute that goes through all the attributes of the media section and returns the direction, which is either "sendrecv", "sendonly", "recvonly" or "inactive". If none of these attributes is present, the default direction is "sendrecv".

SDPConnection

The SDPConnection and FrozenSDPConnection objects represents the contents of a "c" (connection) line of a SDP body, either at the session level or for an individual media stream. It is a simple wrapper for the PJSIP pjmedia_sdp_conn structure. A (Frozen)SDPConnection object can be contained in a (Frozen)SDPSession object or (Frozen)SDPMediaStream object. It supports comparison to other (Frozen)SDPConnection objects using the == and != expressions. As all the attributes of this class are set by attributes of the __init__ method, they will be documented along with that method.

methods

```
__init__(self, address, net_type="IN", address_type="IP4")
```

Creates the SDPConnection object with the specified parameters as attributes. Each of these attributes can be accessed and changed on the object once instanced.

address:

The address part of the connection line as a string.

net_type:

The network type part of the connection line as a string.

address_type:

The address type part of the connection line as a string.

```
new(cls, sdp_connection)
```

Classmethod that returns an instance of the class on which it has been called which is a copy of the sdp_connection object (which must be either a SDPConnection or a FrozenSDPConnection).

SDPAttributeList

SDPAttributeList and FrozenSDPAttributeList are subclasses of list and frozenlist respectively and are used as the types of the attributes attributes of (Frozen)SDPSession and (Frozen)SDPMediaStream. They provide convinience methods for accessing SDP attributes. Apart from the standard list and frozenlist methods, they also provide the following:

```
__contains__(self, item)
```

If item is an instance of BaseSDPAttribute, the normal (frozen)list method is called. Otherwise, the method returns whether or not item is in the list of the names of the attributes. This allows tests such as the following to be possible:

```
'ice-pwd' in sdp_session.attributes
```

```
getall(self, name)
```

Returns all the values of the attributes with the given name in a list.

```
getfirst(self, name, default=None)
```

Return the first value of the attribute with the given name, or default is no such attribute exists.

SDPAttribute

The SDPAttribute and FrozenSDPAttribute objects represent the contents of a "a" (attribute) line of a SDP body, either at the session level or for an individual media stream. It is a simple wrapper for the PJSIP pjmedia_sdp_attr structure. One or more (Frozen)SDPAttribute objects can be contained in a (Frozen)SDPSession object or (Frozen)SDPMediaStream object. It supports comparison to other (Frozen)SDPAttribute objects using the == and != expressions. As all the attributes of this class are set by attributes of the __init__ method, they will be documented along with that method.

methods

```
__init__(self, name, value)
```

Creates the SDPAttribute object with the specified parameters as attributes. Each of these attributes can be accessed and changed on the object once instanced.

name:

The name part of the attribute line as a string.

value:

The value part of the attribute line as a string.

```
new(cls, sdp_attribute)
```

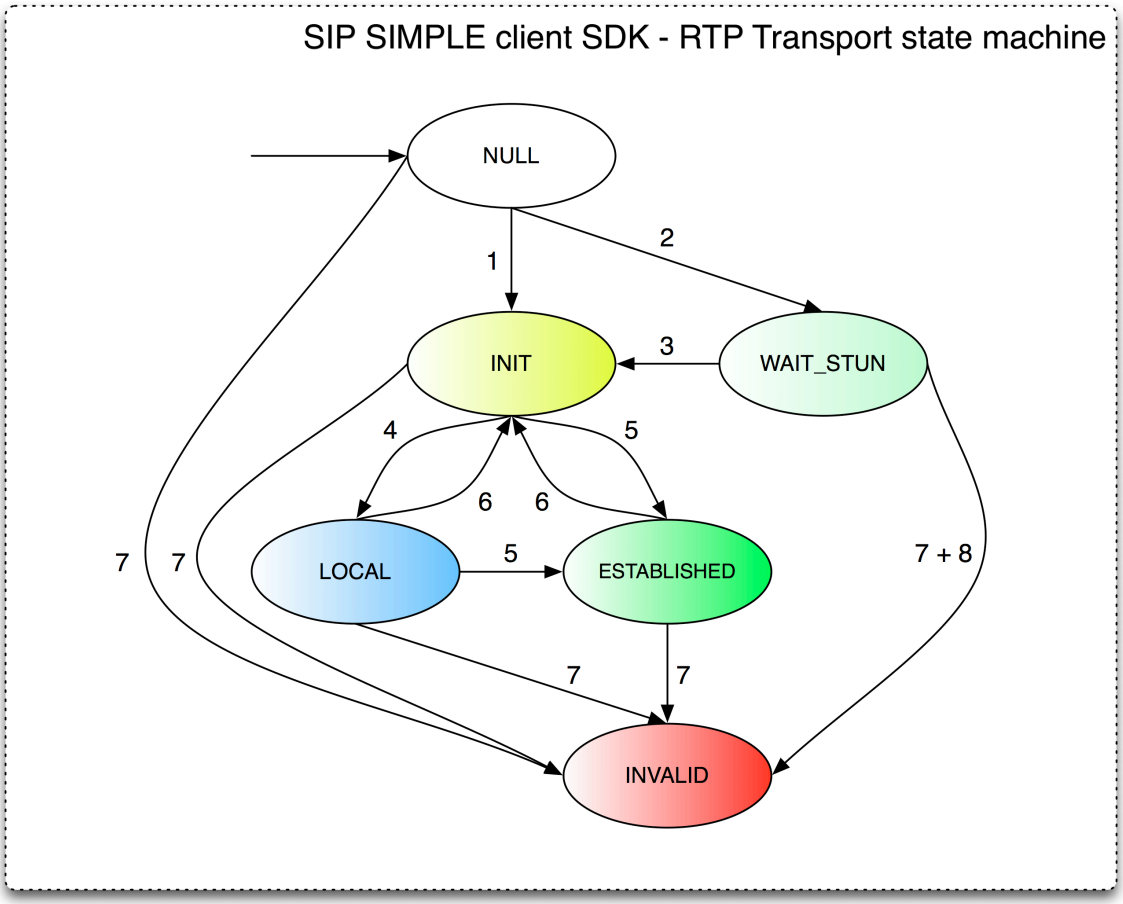
Classmethod that returns an instance of the class on which it has been called which is a copy of the sdp_attribute object (which must be either a SDPAttribute or a FrozenSDPAttribute).

RTPTransport

This object represents a transport for RTP media, the basis of which is a pair of UDP sockets, one for RTP and one for RTCP. Internally it wraps a pjmedia_transport object. Initially this object will only be used by the AudioTransport object, but in the future it can also be used for video and Real-Time Text?. For this reason the AudioTransport and RTPTransport are two distinct objects.

The RTPTransport object also allows support for ICE and SRTP functionality from PJSIP. Because these features are related to both the UDP transport and the SDP formatting, the SDP carried in SIP signaling message will need to "pass through" this object during the SDP negotiation. The code using this object, which in most cases will be the AudioTransport object, will need to call certain methods on the object at appropriate times. This process of SDP negotiation is represented by the internal state machine of the object, as shown in the following diagram:

The Real-time Transport Protocol (or RTP) defines a standardized packet format for delivering audio and video over the Internet. It was developed by the Audio-Video Transport Working Group of the IETF and published in RFC 3550. RTP is used in streaming media systems (together with the RTSP) as well as in videoconferencing and push to talk systems. For these it carries media streams controlled by Session Initiation Protocol (SIP) signaling protocols, making it the technical foundation of the Voice over IP industry.



State changes are triggered by the following events:

1. The application calls the set_INIT() method after object creation and ICE+STUN is not used.
2. The application calls the set_INIT() method after object creation and ICE+STUN is used.
3. A successful STUN response is received from the STUN server.
4. The set_LOCAL() method is called.
5. The set_ESTABLISHED() method is called.
6. The set_INIT() method is called while the object is in the LOCAL or ESTABLISHED state.

A method is called on the application, but in the meantime the Engine has stopped. The object can no longer be used.

There was an error in getting the STUN candidates from the STUN server.

It would make sense to be able to use the object even if the STUN request fails (and have ICE not include a STUN candidate), but for some reason the pjmedia_transport is unusable once STUN negotiation has failed. This means that the RTPTransport object is also unusable once it has reached the STUN_FAILED state. A workaround would be destroy the RTPTransport object and create a new one that uses ICE without STUN.

These states allow for two SDP negotiation scenarios to occur, represented by two paths that can be followed through the state machine. In this example we will assume that ICE with STUN is not used, as it is independent of the SDP negotiation procedure.

The first scenario is where the local party generates the SDP offer. For a stream that it wishes to include in this SDP offer, it instantiates a RTPTransport object. After instantiation the object is initialized by calling the set_INIT() method and the local RTP address and port can be fetched from it using the local_rtp_address and local_rtp_port attributes respectively, which can be used to generate the local SDP in the form of a SDPSession object. This local SDP then needs to be passed to the set_LOCAL() method, which moves the state machine into the LOCAL state (note that it needs the full object, not just the relevant SDPMediaStream object). Depending on the options used for the RTPTransport instantiation (such as ICE and SRTP), this may change the SDPSession object. This (possibly changed) SDPSession object then needs to be passed to the Invitation object. After SDP negotiation is completed, the application needs to pass both the local and remote SDP, in the form of (Frozen)SDPSession objects, to the RTPTransport object using the set_ESTABLISHED() method, moving the state machine into the ESTABLISHED state. This will not change either of the (Frozen)SDPSession objects (which is why they can also be frozen).

The second scenario is where the local party is offered a media stream in SDP and wants to accept it. In this case a RTPTransport is also instantiated and initialized using the set_INIT() method, and the application can generate the local SDP in response to the remote SDP, using the local_rtp_address and local_rtp_port attributes. Directly after this it should pass the generated local SDP and received remote SDP, in the form of SDPSession objects, to the set_ESTABLISHED() method. In this case the local SDP object may be changed, after which it can be passed to the Invitation object.

Whenever the RTPTransport object is in the LOCAL or ESTABLISHED states, it may be reset to the INIT state to facilitate re-use of the existing transport and its features. Before doing this however, the internal transport object must no longer be in use.

methods

```
__init__(self, local_rtp_address=None, use_srtp=False,
srtp_forced=False, use_ice=False, ice_stun_address=None,
ice_stun_port=3478)
```

Creates a new RTPTransport object and opens the RTP and RTCP UDP sockets. If additional features are requested, they will be initialized. After object instantiation, it is either in the INIT or the WAIT_STUN state, depending on the values of the use_ice and ice_stun_address arguments.

local_rtp_address:

Optionally contains the local IPv4 address to listen on. If this is not specified, PJSIP will listen on all network interfaces.

use_srtp:

A boolean indicating if SRTP should be used. If this is set to True, SRTP information will be added to the SDP when it passes this object.

srtp_forced:

A boolean indicating if use of SRTP is set to mandatory in the SDP. If this is set to True and the remote party does not support SRTP, the SDP negotiation for this stream will fail. This argument is relevant only if use_srtp is set to True.

use_ice:

A boolean indicating if ICE should be used. If this is set to True, ICE candidates will be added to the SDP when it passes this object.

ice_stun_address:

A string indicating the address (IP address or hostname) of the STUN server that should be used to add a STUN candidate to the ICE candidates. If this is set to None no STUN candidate will be added, otherwise the object will be put into the WAIT_STUN state until a reply, either positive or negative, is received from the specified STUN server. When this happens a RTPTransportGotSTUNResponse notification is sent. This argument is relevant only if use_ice is set to True.

ice_stun_address:

An int indicating the UDP port of the STUN server that should be used to add a STUN candidate to the ICE candidates. This argument is relevant only if use_ice is set to True and ice_stun_address is not None.

```
set_INIT(self)
```

This moves the internal state machine into the INIT state. If the state machine is in the LOCAL or ESTABLISHED states, this effectively resets the RTPTransport object for re-use.

```
set_LOCAL(self, local_sdp, sdp_index)
```

This moves the the internal state machine into the LOCAL state.

local_sdp:

The local SDP to be proposed in the form of a SDPSession object. Note that this object may be modified by this method.

sdp_index:

The index in the SDP for the media stream for which this object was created.

```
set_ESTABLISHED(self, local_sdp, remote_sdp, sdp_index)
```

This moves the the internal state machine into the ESTABLISHED state.

local_sdp:

The local SDP to be proposed in the form of a SDPSession object. Note that this object may be modified by this method, but only when moving from the LOCAL to the ESTABLISHED state.

remote_sdp:

The remote SDP that was received in in the form of a SDPSession object.

sdp_index:

The index in the SDP for the media stream for which this object was created.

attributes

```
state
```

Indicates which state the internal state machine is in. See the previous section for a list of states the state machine can be in. This attribute is read-only.

```
local_rtp_address
```

The local IPv4 address of the interface the RTPTransport object is listening on and the address that should be included in the SDP. If no address was specified during object instantiation, PJSIP will take guess out of the IP addresses of all interfaces. This attribute is read-only and will be None if PJSIP is not listening on the transport.

```
local_rtp_port
```

The UDP port PJSIP is listening on for RTP traffic. RTCP traffic will always be this port plus one. This attribute is read-only and will be None if PJSIP is not listening on the transport.

`remote_rtp_address_sdp`

The remote IP address that was seen in the SDP. This attribute is read-only and will be None unless the object is in the ESTABLISHED state.

`remote_rtp_port_sdp`

The remote UDP port for RTP that was seen in the SDP. This attribute is read-only and will be None unless the object is in the ESTABLISHED state.

`remote_rtp_address_ice`

The remote IP address that was selected by the ICE negotation. This attribute is read-only and will be None until the ICE negotation succeeds.

`remote_rtp_port_ice`

The remote port that was selected by the ICE negotation. This attribute is read-only and will be None until the ICE negotation succeeds.

`remote_rtp_address_received`

The remote IP address from which RTP data was received. This attribute is read-only and will be None unless RTP was actually received.

`remote_rtp_port_received`

The remote UDP port from which RTP data was received. This attribute is read-only and will be None unless RTP was actually received.

`use_srtp`

A boolean indicating if the use of SRTP was requested when the object was instantiated. This attribute is read-only.

`force_srtp`

A boolean indicating if SRTP being mandatory for this transport if it is enabled was requested when the object was instantiated. This attribute is read-only.

`srtp_active`

A boolean indicating if SRTP encryption and decryption is running. Querying this attribute only makes sense once the object is in the ESTABLISHED state and use of SRTP was requested. This attribute is read-only.

```
use_ice
```

A boolean indicating if the use of ICE was requested when the object was instantiated. This attribute is read-only.

ice_active

A boolean indicating if ICE is being used. This attribute is read-only.

```
ice_stun_address
```

A string indicating the IP address of the STUN server that was requested to be used. This attribute is read-only.

```
ice_stun_port
```

A string indicating the UDP port of the STUN server that was requested to be used. This attribute is read-only.

```
local_rtp_candidate_type
```

Returns the ICE candidate type which has been selected for the local endpoint.

```
remote_rtp_candidate_type
```

Returns the ICE candidate type which has been selected for the remote endpoint.

notifications

```
RTPTransportDidInitialize
```

This notification is sent when a RTPTransport object has successfully initialized. If STUN+ICE is not requested, this is sent immediately on set_INIT(), otherwise it is sent after the STUN query has succeeded.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
RTPTransportDidFail
```

This notification is sent by a RTPTransport object that fails to retrieve ICE candidates from the STUN server after set_INIT() is called.

timestamp:

A datetime.datetime object indicating when the notification was sent.

reason:

A string describing the failure reason.

```
RTPTransportICENegotiationStateDidChange
```

This notification is sent to indicate the progress of the ICE negotiation.

state:

A string describing the current ICE negotiation state.

```
RTPTransportICENegotiationDidFail
```

This notification is sent when the ICE negotiation fails.

reason:

A string describing the failure reason of ICE negotation.

```
RTPTransportICENegotiationDidSucceed
```

This notification is sent when the ICE negotation succeeds.

chosen_local_candidates and chosen_remote_candidates:

Dictionaries with the following keys:

- rtp_cand_type: the type of the RTP candidate
- rtp_cand_ip: the IP address of the RTP candidate
- rtcp_cand_type: the type of the RTCP candidate
- rtcp_cand_ip: the IP address of teh RTCP candidate

duration:

The amount of time the ICE negotiation took.

local_candidates and remote_candidates:

Lists of tuples with the following elements:

- Item ID
- Component ID
- Address
- Component Type

connectivity_checks_results:

A list of tuples with the following elements:

- Item ID
- Component ID
- Source
- Destination
- Nomination
- State

AudioTransport

This object represent an audio stream as it is transported over the network. It contains an instance of RTPTransport and wraps a pjmedia_stream object, which in turn manages the RTP encapsulation, RTCP session, audio codec and adaptive jitter buffer. It also generates a SDPMediaStream object to be included in the local SDP.

The AudioTransport is an object that, once started, is connected to a AudioMixer instance, and both produces and consumes sound.

Like the RTPTransport object there are two usage scenarios.

In the first scenario, only the RTPTransport instance to be used is passed to the AudioTransport object. The application can then generate the SDPMediaStream object by calling the get_local_media() method and should include it in the SDP offer. Once the remote SDP is received, it should be set along with the complete local SDP by calling the start() method, which will start the audio stream. The stream can then be connected to the conference bridge.

In the other scenario the remote SDP is already known because it was received in an SDP offer and can be passed directly on object instantiation. The local SDPMediaStream object can again be generated by calling the get_local_media() method and is to be included in the SDP answer. The audio stream is started directly when the object is created.

Unlike the RTPTransport object, this object cannot be reused.

methods

```
__init__(self, mixer, transport, remote_sdp=None, sdp_index=0,
enable_silence_detection=True, codecs=None)
```

Creates a new AudioTransport object and start the underlying stream if the remote SDP is already known.

mixer:

The AudioMixer object that this object is to be connected to.

transport:

The transport to use in the form of a RTPTransport object.

remote_sdp:

The remote SDP that was received in the form of a SDPSession object.

sdp_index:

The index within the SDP of the audio stream that should be created.

enable_silence_detection:

Boolean that indicates if silence detection should be used for this audio stream. When enabled, this AudioTransport object will stop sending audio to the remote party if the input volume is below a certain threshold.

Codecs:

A list of strings indicating the codecs that should be proposed in the SDP of this AudioTransport, in order of preference. This overrides the global codecs list set on the Engine. The values of this list are case insensitive.

```
get_local_media(self, is_offer, direction="sendrecv")
```

Generates a SDPMediaStream object which describes the audio stream. This object should be included in a SDPSession object that gets passed to the Invitation object. This method should also be used to obtain the SDP to include in re-INVITEs and replies to re-INVITEs.

is_offer:

A boolean indicating if the SDP requested is to be included in an offer. If this is False it is to be included in an answer.

direction:

The direction attribute to put in the SDP.

```
start(self, local_sdp, remote_sdp, sdp_index,
no_media_timeout=10, media_check_interval=30)
```

This method should only be called once, when the application has previously sent an SDP offer and the answer has been received.

local_sdp:

The full local SDP that was included in the SDP negotiation in the form of a SDPSession object.

remote_sdp:

The remote SDP that was received in the form of a SDPSession object.

sdp_index:

The index within the SDP of the audio stream.

no_media_timeout:

This argument indicates after how many seconds after starting the AudioTransport the RTPAudioTransportDidNotGetRTP notification should be sent, if no RTP has been received at all. Setting this to 0 disables this an all subsequent RTP checks.

media_check_interval:

This indicates the interval at which the RTP stream should be checked, after it has initially received RTP at after no_media_timeout seconds. It means that if between two of these interval checks no RTP has been received, a RTPAudioTransportDidNotGetRTP notification will be sent. Setting this to 0 will disable checking the RTP at intervals. The initial check may still be performed if its timeout is non-zero.

```
stop(self)
```

This method stops and destroys the audio stream encapsulated by this object. After this it can no longer be used and should be deleted, while the RTPTransport object used by it can be re-used for something else. This method will be called automatically when the object is deleted after it was started, but this should not be relied on because of possible reference counting issues.

```
send_dtmf(self, digit)
```

For a negotiated audio transport this sends one DTMF digit to the other party

digit:

A string of length one indicating the DTMF digit to send. This can be either a digit, the pound sign (#), the asterisk sign (*) or the letters A through D.

```
update_direction(self, direction)
```

This method should be called after SDP negotiation has completed to update the direction of the media stream.

direction:

The direction that has been negotiated.

attributes

`mixer`

The AudioMixer object that was passed when the object got instantiated. This attribute is read-only.

`transport`

The RTPTransport object that was passed when the object got instantiated. This attribute is read-only.

`slot`

A read-only property indicating the slot number at which this object is attached to the associated conference bridge. If the AudioTransport is not active (i.e. has not been started), this attribute will be None.

`volume`

A writable property indicating the % of volume at which this object contributes audio to the conference bridge. By default this is set to 100.

`is_active`

A boolean indicating if the object is currently sending and receiving audio. This attribute is read-only.

`is_started`

A boolean indicating if the object has been started. Both this attribute and the is_active attribute get set to True once the start() method is called, but unlike the is_active attribute this attribute does not get set to False once stop() is called. This is to prevent the object from being re-used. This attribute is read-only.

`codec`

Once the SDP negotiation is complete, this attribute indicates the audio codec that was negotiated, otherwise it will be None. This attribute is read-only.

`sample_rate`

Once the SDP negotiation is complete, this attribute indicates the sample rate of the audio codec that was negotiated, otherwise it will be None. This attribute is read-only.

```
direction
```

The current direction of the audio transport, which is one of "sendrecv", "sendonly", "recvonly" or "inactive". This attribute is read-only, although it can be set using the update_direction() method.

notifications

```
RTPAudioTransportGotDTMF
```

This notification will be sent when an incoming DTMF digit is received from the remote party.

timestamp:

A datetime.datetime object indicating when the notification was sent.

digit:

The DTMF digit that was received, in the form of a string of length one. This can be either a number or letters A through D.

```
RTPAudioTransportDidNotGetRTP
```

This notification will be sent when no RTP packets have been received from the remote party for some time. See the start() method for a more exact description.

timestamp:

A datetime.datetime object indicating when the notification was sent.

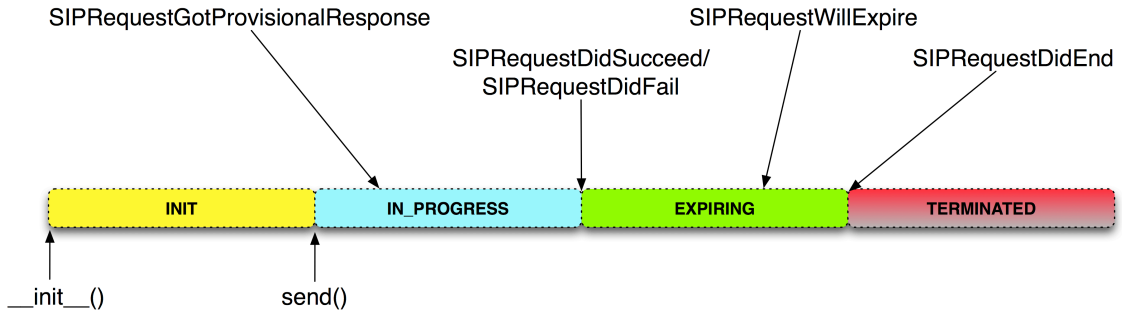
got_any:

A boolean data attribute indicating if the AudioTransport every saw any RTP packets from the remote party. In effect, if no RTP was received after no_media_timeout seconds, its value will be False.

Request

The sipsimple.core.Request object encapsulates a single SIP request transaction from the client side, which includes sending the request, receiving the response and possibly waiting for the result of the request to expire. Although this class can be used by the application to construct and send an arbitrary SIP request, most applications will use the classes for primitive requests provided in the sipsimple.core module, which are built on top of one or several Request objects and deal with instances of specific SIP methods (REGISTER, MESSAGE and PUBLISH).

The lifetime of this object is linear and is described by the following diagram:



The bar denotes which state the object is in and at the top are the notifications which may be emitted at certain points in time. Directly after the object is instantiated, it will be in the INIT state. The request will be sent over the network once its send() method is called, moving the object into the IN_PROGRESS state.

On each provisional response that is received in reply to this request, the SIPRequestGotProvisionalResponse notification is sent, with data describing the response. Note that this may not occur at all if not provisional responses are received. When the send() method has been called and it does not return an exception, the object will send either a SIPRequestDidSucceed or a SIPRequestDidFail notification. Both of these notifications include data on the response that triggered it.

Note that a SIP response that requests authentication (401 or 407) will be handled internally the first time, if a Credentials object was supplied. If this is the sort of request that expires (detected by a Expires header in the response or a expires parameter in the Contact header of the response), and the request was successful, the object will go into the EXPIRING state. A certain amount of time before the result of the request will expire, governed by the expire_warning_time class attribute and the actual returned expiration time, a SIPRequestWillExpire notification will be sent. This should usually trigger whomever is using this Request object to construct a new Request for a refreshing operation.

When the Request actually expires, or when the EXPIRING state is skipped directly after sending SIPRequestDidSucceed or SIPRequestDidFail, a SIPRequestDidEnd notification will be sent.

methods

```
__init__(self, method, request_uri, from_header, to_header,
request_uri, route_header, credentials=None, contact_header=None,
call_id=None, cseq=None, extra_headers=None, content_type=None,
body=None)
```

Creates a new Request object in the INIT state. The arguments to this method are documented in the attributes section.

```
send(self, timeout=None)
```

Compose the SIP request and send it to the destination. This moves the Request object into the IN_PROGRESS state.

timeout:

This can be either an int or a float, indicating in how many seconds the request should timeout with an internally generated 408 response. This is is None, the internal 408 is only triggered by the internal PJSIP transaction timeout. Note that, even if the timeout is specified, the PJSIP timeout is also still valid.

```
end(self)
```

Terminate the transaction, whichever state it is in, sending the appropriate notifications. Note that calling this method while in the INIT state does nothing.

attributes

```
expire_warning_time (class attribute)
```

The SIPRequestWillExpire notification will be sent halfway between the positive response and the actual expiration time, but at least this amount of seconds before. The default value is 30 seconds.

```
state
```

Indicates the state the Request object is in, in the form of a string. Refer to the diagram above for possible states. This attribute is read-only.

```
method
```

The method of the SIP request as a string. This attribute is set on instantiation and is read-only.

```
from_header
```

The FrozenFromHeader object to put in the From header of the request. This attribute is set on instantiation and is read-only.

`to_header`

The FrozenToHeader object to put in the To header of the request. This attribute is set on instantiation and is read-only.

`request_uri`

The SIP URI to put as request URI in the request, in the form of a FrozenSIPURI object. This attribute is set on instantiation and is read-only.

`route_header`

Where to send the SIP request to, including IP, port and transport, in the form of a FrozenRouteHeader object. This will also be included in the Route header of the request. This attribute is set on instantiation and is read-only.

`credentials`

The credentials to be used when challenged for authentication, represented by a FrozenCredentials object. If no credentials were supplied when the object was created this attribute is None. This attribute is set on instantiation and is read-only.

`contact_header`

The FrozenContactHeader object to put in the Contact header of the request. If this was not specified, this attribute is None. It is set on instantiation and is read-only.

`call_id`

The Call-ID to be used for this transaction as a string. If no call id was specified on instantiation, this attribute contains the generated id. This attribute is set on instantiation and is read-only.

`cseq`

The sequence number to use in the request as an int. If no sequence number was specified on instantiation, this attribute contains the generated sequence number. Note that this number may be increased by one if an authentication challenge is received and a Credentials object is given. This attribute is read-only.

`extra_headers`

Extra headers to include in the request as a frozenlist of header objects. This attribute is set on instantiation and is read-only.

`content_type`

What string to put as content type in the Content-Type headers, if the request contains a body. If no body was specified, this attribute is None It is set on instantiation and is read-only.

```
body
```

The body of the request as a string. If no body was specified, this attribute is None It is set on instantiation and is read-only.

```
expires_in
```

This read-only property indicates in how many seconds from now this Request will expire, if it is in the EXPIRING state. If this is not the case, this property is 0.

```
peer_address
```

This read-only attribute contains the remote endpoint IP and port information. It can be accessed by accessing this object's ip and port attributes.

notifications

```
SIPRequestGotProvisionalResponse
```

This notification will be sent on every provisional response received in reply to the sent request.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The SIP response code of the received provisional response as an int, which will be in the 1xx range.

reason:

The reason text string included with the SIP response code.

headers:

The headers included in the provisional response as a dict, the values of which are header objects.

body:

The body of the provisional response as a string, or None if there was no body.

```
SIPRequestDidSucceed
```

This notification will be sent when a positive final response was received in reply to the request.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The SIP response code of the received positive final response as an int, which will be in the 2xx range.

reason:

The reason text string included with the SIP response code.

headers:

The headers included in the positive final response as a dict, the values of which are header objects.

body:

The body of the positive final response as a string, or None if there was no body.

expires:

Indicates in how many seconds the Request will expire as an int. If it does not expire and the EXPIRING state is skipped, this attribute is 0.

```
SIPRequestDidFail
```

This notification will be sent when a negative final response is received in reply to the request or if an internal error occurs.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The SIP response code of the received negative final response as an int. This could also be a response code generated by PJSIP internally, or 0 when an internal error occurs.

reason:

The reason text string included with the SIP response code or error.

headers: (only if a negative final response was received)

The headers included in the negative final response as a dict, the values of which are header objects, if this is what triggered the notification.

body: (only if a negative final response was received)

The body of the negative final response as a string, or None if there was no body. This attribute is absent if no response was received.

```
SIPRequestWillExpire
```

This notification will be sent when a Request in the EXPIRING state will expire soon.

timestamp:

A datetime.datetime object indicating when the notification was sent.

expires:

An int indicating in how many seconds from now the Request will actually expire.

```
SIPRequestDidEnd
```

This notification will be sent when a Request object enters the TERMINATED state and can no longer be used.

timestamp:

A datetime.datetime object indicating when the notification was sent.

IncomingRequest

This is a relatively simple object that can manage responding to incoming requests in a single transaction. For this reason it does not handle requests that create a dialog. To register methods for which requests should be formed into an IncomingRequest object, the application should set the incoming_requests set attribute on the Engine. Receiving INVITE, SUBSCRIBE, ACK and BYE through this object is not supported.

The application is notified of an incoming request through the SIPIncomingRequestGotRequest notification. It can answer this request by calling the answer method on the sender of this notification. Note that if the IncomingRequest object gets destroyed before it is answered, a 500 response is automatically sent.

attributes

```
state
```

This read-only attribute indicates the state the object is in. This can be None if the object was created by the application, essentially meaning the object is inert, "incoming" or "answered".

methods

```
answer(self, code, reason=None, extra_headers=None)
```

Reply to the received request with a final response.

code:

The SIP response code to send. This should be 200 or higher.

reason:

The reason text to include in the response. If this is None, the default text for the given response code is used.

extra_headers:

The extra headers to include in the response as an iterable of header objects.

notifications

```
SIPIncomingRequestGotRequest
```

This notification will be sent when a new IncomingRequest is created as result of a received request. The application should listen for this notification, retain a reference to the object that sent it and answer it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

method:

The method of the SIP request as a string.

request_uri:

The request URI of the request as a FrozenSIPURI object.

headers:

The headers of the request as a dict, the values of which are the relevant header objects.

body:

The body of the request as a string, or None if no body was included.

```
SIPIncomingRequestSentResponse
```

This notification is sent when a response to the request is sent by the object.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The code of the SIP response as an int.

reason:

The reason text of the SIP response as an int.

headers:

The headers of the response as a dict, the values of which are the relevant header objects.

body:

This will be None.

Message

The Message class is a simple wrapper for the Request class, with the purpose of sending MESSAGE requests, as described in RFC 3428. It is a one-shot object that manages only one Request object.

methods

```
__init__(self, from_header, to_header, route_header,
content_type, body, credentials=None, extra_headers=[])
```

Creates a new Message object with the specified arguments. These arguments are documented in the attributes section for this class.

```
send(self, timeout=None)
```

Send the MESSAGE request to the remote party.

timeout:

This argument as the same meaning as it does for Request.send().

```
end(self)
```

Stop trying to send the MESSAGE request. If it was not sent yet, calling this method does nothing.

attributes

```
from_header
```

The FrozenFromHeader to put in the From header of the MESSAGE request. This attribute is set on instantiation and is read-only.

```
to_header
```

The FrozenToHeader to put in the To header of the MESSAGE request. This attribute is set on instantiation and is read-only.

```
route_header
```

Where to send the MESSAGE request to, including IP, port and transport, in the form of a FrozenRouteHeader object. This will also be included in the Route header of the request. This attribute is set on instantiation and is read-only.

`content_type`

What string to put as content type in the Content-Type headers. It is set on instantiation and is read-only.

`body`

The body of the MESSAGE request as a string. If no body was specified, this attribute is None It is set on instantiation and is read-only.

`credentials`

The credentials to be used when challenged for authentication, represented by a FrozenCredentials object. If no credentials were specified, this attribute is None. This attribute is set on instantiation and is read-only.

`is_sent`

A boolean read-only property indicating if the MESSAGE request was sent.

`in_progress`

A boolean read-only property indicating if the object is waiting for the response from the remote party.

`peer_address`

This read-only attribute contains the remote endpoint IP and port information. It can be accessed by accessing this object's ip and port attributes.

notifications

`SIPMessageDidSucceed`

This notification will be sent when the remote party acknowledged the reception of the MESSAGE request. It has not data attributes.

`SIPMessageDidFail`

This notification will be sent when transmission of the MESSAGE request fails for whatever reason.

code:

An int indicating the SIP or internal PJSIP code that was given in response to the MESSAGE request. This is 0 if the failure is caused by an internal error.

reason:

```
A status string describing the failure, either taken from the SIP
response or the internal error.
```

Registration

The Registration class wraps a series of Request objects, managing the registration of a particular contact URI at a SIP registrar through the sending of REGISTER requests. For details, see RFC 3261. This object is designed in such a way that it will not initiate sending a refreshing REGISTER itself, rather it will inform the application that a registration is about to expire. The application should then perform a DNS lookup to find the relevant SIP registrar and call the register() method on this object.

methods

```
__init__(self, from_header, credentials=None, duration=300)
```

Creates a new Registration object with the specified arguments. These arguments are documented in the attributes section for this class.

```
register(self, contact_header, route_header, timeout=None)
```

Calling this method will attempt to send a new REGISTER request to the registrar provided, whatever state the object is in. If the object is currently busy sending a REGISTER, this request will be preempted for the new one. Once sent, the Registration object will send either a SIPRegistrationDidSucceed or a SIPRegistrationDidFail notification. The contact_header argument is mentioned in the attributes section of this class. The route_header argument specifies the IP address, port and transport of the SIP registrar in the form of a RouteHeader object and timeout has the same meaning as it does for Request.send().

```
end(self, timeout=None)
```

Calling this method while the object is registered will attempt to send a REGISTER request with the Expires headers set to 0, effectively unregistering the contact URI at the registrar. The RouteHeader used for this will be the same as the last successfully sent REGISTER request. If another REGISTER is currently being sent, it will be preempted. When the unregistering REGISTER request is sent, a SIPRegistrationWillEnd notification is sent. After this, either a SIPRegistrationDidEnd with the expired data attribute set to False will be sent, indicating a successful unregister, or a SIPRegistrationDidNotEnd notification is sent if unregistering fails for some reason. Calling this method when the object is not registered will do nothing. The timeout argument has the same meaning as for the register() method.

attributes

```
from_header
```

The (Frozen)FromHeader the Registration was sent with.

```
credentials
```

The credentials to be used when challenged for authentication by the registrar, represented by a (Frozen)Credentials object. This attribute is set at instantiation, can be None if it was not specified and can be updated to be used for the next REGISTER request. Note that, in contrast to other classes mentioned in this document, the Registration class does not make a copy of the Credentials object on instantiation, but instead retains a reference to it.

```
duration
```

The amount of time in seconds to request the registration for at the registrar. This attribute is set at object instantiation and can be updated for subsequent REGISTER requests.

```
is_registered
```

A boolean read-only property indicating if this object is currently registered.

contact_header

If the Registration object is registered, this attribute contains the latest contact header that was sent to the registrar as a FrozenContactHeader object. Otherwise, this attribute is None

```
expires_in
```

If registered, this read-only property indicates in how many seconds from now this Registration will expire. If this is not the case, this property is 0.

notifications

```
SIPRegistrationDidSucceed
```

This notification will be sent when the register() method was called and the registration succeeded.

code:

The SIP response code as received from the registrar as an int.

reason:

```
The reason string received from the SIP registrar.
```

route_header:

The (Frozen)RouteHeader object passed as an argument to the register() method.

contact_header:

The contact header that was sent to the registrar as a FrozenContactHeader object.

contact_header_list:

A full list of contact headers that are registered for this SIP account, including the one used for this registration. The format of this data attribute is a list of FrozenContactHeader objects.

expires_in:

The number of seconds before this registration expires

```
SIPRegistrationDidFail
```

This notification will be sent when the register() method was called and the registration failed, for whatever reason.

code:

The SIP response code as received from the registrar as an int. This can also be a PJSIP generated response code, or 0 if the failure was because of an internal error.

reason:

The reason string received from the SIP registrar or the error string.

route_header:

The (Frozen)RouteHeader object passed as an argument to the register() method.

```
SIPRegistrationWillExpire
```

This notification will be sent when a registration will expire soon. It should be used as an indication to re-perform DNS lookup of the registrar and call the register() method.

expires:

The number of seconds in which the registration will expire.

```
SIPRegistrationWillEnd
```

Will be sent whenever the end() method was called and an unregistering REGISTER is sent.

```
SIPRegistrationDidNotEnd
```

This notification will be sent when the unregistering REGISTER request failed for some reason.

code:

The SIP response code as received from the registrar as an int. This can also be a PJSIP generated response code, or 0 if the failure was because of an internal error.

reason:

The reason string received from the SIP registrar or the error string.

```
SIPRegistrationDidEnd
```

This notification will be sent when a Registration has become unregistered.

expired:

This boolean indicates if the object is unregistered because the registration expired, in which case it will be set to True. If this data attribute is False, it means that unregistration was explicitly requested through the end() method.

Publication

Publication of SIP events is an Internet standard published at RFC 3903. PUBLISH is similar to REGISTER in that it allows a user to create, modify, and remove state in another entity which manages this state on behalf of the user.

A Publication object represents publishing some content for a particular SIP account and a particular event type at the SIP presence agent through a series of PUBLISH request. This object is similar in behaviour to the Registration object, as it is also based on a sequence of Request objects. As with this other object, the Publication object will notify the application when a published item is about to expire and it is the applications responsibility to perform a DNS lookup and call the publish() method in order to send the PUBLISH request.

methods

```
__init__(self, from_header, event, content_type,
credentials=None, duration=300)
```

Creates a new Publication object with the specified arguments. These arguments are documented in the attributes section for this class.

```
publish(self, body, route_header, timeout=None)
```

Send an actual PUBLISH request to the specified presence agent.

body:

The body to place in the PUBLISH request as a string. This body needs to be of the content type specified at object creation. For the initial request, a body will need to be specified. For subsequent refreshing PUBLISH requests, the body can be None to indicate that the last published body should be refreshed. If there was an ETag error with the previous refreshing PUBLISH, calling this method with a body of None will throw a PublicationError.

route_header:

The host where the request should be sent to in the form of a (Frozen)RouteHeader object.

timeout:

This can be either an int or a float, indicating in how many seconds the SUBSCRIBE request should timeout with an internally generated 408 response. This is is None, the internal 408 is only triggered by the internal PJSIP transaction timeout. Note that, even if the timeout is specified, the PJSIP timeout is also still valid.

```
end(self, timeout=None)
```

End the publication by sending a PUBLISH request with an Expires header of 0. If the object is not published, this will result in a PublicationError being thrown.

timeout:

The meaning of this argument is the same as it is for the publish() method.

attributes

`from_header`

The (Frozen)FromHeader the Publication was sent with.

`event`

The name of the event this object is publishing for the specified SIP URI, as a string.

`content_type`

The Content-Type of the body that will be in the PUBLISH requests. Typically this will remain the same throughout the publication session, but the attribute may be updated by the application if needed. Note that this attribute needs to be in the typical MIME type form, containing a "/" character.

`credentials`

The credentials to be used when challenged for authentication by the presence agent, represented by a (Frozen)Credentials object. This attribute is set at instantiation, can be None if it was not specified and can be updated to be used for the next PUBLISH request. Note that, in contrast to most other classes mentioned in this document, the Publication class does not make a copy of the (Frozen)Credentials object on instantiation, but instead retains a reference to it.

`duration`

The amount of time in seconds that the publication should be valid for. This attribute is set at object instantiation and can be updated for subsequent PUBLISH requests.

`is_published`

A boolean read-only property indicating if this object is currently published.

`expires_in`

If published, this read-only property indicates in how many seconds from now this Publication will expire. If this is not the case, this property is 0.

notifications

`SIPPublicationDidSucceed`

This notification will be sent when the publish() method was called and the publication succeeded.

code:

The SIP response code as received from the SIP presence agent as an int.

reason:

The reason string received from the SIP presence agent.

route_header:

The (Frozen)RouteHeader object passed as an argument to the publish() method.

expires_in:

The number of seconds before this publication expires

```
SIPPublicationDidFail
```

This notification will be sent when the publish() method was called and the publication failed, for whatever reason.

code:

The SIP response code as received from the presence agent as an int. This can also be a PJSIP generated response code, or 0 if the failure was because of an internal error.

reason:

The reason string received from the presence agent or the error string.

route_header:

The (Frozen)RouteHeader object passed as an argument to the publish() method.

```
SIPPublicationWillExpire
```

This notification will be sent when a publication will expire soon. It should be used as an indication to re-perform DNS lookup of the presence agent and call the publish() method.

expires:

The number of seconds in which the publication will expire.

```
SIPPublicationWillEnd
```

Will be sent whenever the end() method was called and an unpublishing PUBLISH is sent.

```
SIPPublicationDidNotEnd
```

This notification will be sent when the unpublishing PUBLISH request failed for some reason.

code:

The SIP response code as received from the presence agent as an int. This can also be a PJSIP generated response code, or 0 if the failure was because of an internal error.

reason:

The reason string received from the presence agent or the error string.

```
SIPPublicationDidEnd
```

This notification will be sent when a Publication has become unpublished.

expired:

This boolean indicates if the object is unpublished because the publication expired, in which case it will be set to True. If this data attribute is False, it means that unpublication was explicitly requested through the end() method.

Subscription

Subscription and notifications for SIP events is an Internet standard published at RFC 3856.

This SIP primitive class represents a subscription to a specific event type of a particular SIP URI. This means that the application should instance this class for each combination of event and SIP URI that it wishes to subscribe to. The event type and the content types that are acceptable for it need to be registered first, either through the init_options attribute of Engine (before starting it), or by calling the add_event() method of the Engine instance. Whenever a NOTIFY is received, the application will receive the SIPSubscriptionGotNotify event.

Internally a Subscription object has a state machine, which reflects the state of the subscription. It is a direct mirror of the state machine of the underlying pjsip_evsub object, whose possible states are at least NULL, SENT, ACCEPTED, PENDING, ACTIVE or TERMINATED. The last three states are directly copied from the contents of the Subscription-State header of the received NOTIFY request. Also, the state can be an arbitrary string if the contents of the Subscription-State header are not one of the three above. The state of the object is presented in the state attribute and changes of the state are indicated by means of the SIPSubscriptionChangedState notification. This notification is only informational, an application using this object should take actions based on the other notifications sent by this object.

methods

```
__init__(self, request_uri, from_header, to_header,
contact_header, event, route_header, credentials=None,
refresh=300)
```

Creates a new Subscription object which will be in the NULL state. The arguments for this method are documented in the attributes section above.

```
subscribe(self, extra_headers=None, content_type=None, body=None,
timeout=None)
```

Calling this method triggers sending a SUBSCRIBE request to the presence agent. The arguments passed will be stored and used for subsequent refreshing SUBSCRIBE requests. This method may be used both to send the initial request and to update the arguments while the subscription is ongoing. It may not be called when the object is in the TERMINATED state.

extra_headers:

A dictionary of additional headers to include in the SUBSCRIBE requests.

content_type:

The Content-Type of the supplied body argument as a string. If this argument is supplied, the body argument cannot be None.

body:

The optional body to include in the SUBSCRIBE request as a string. If this argument is supplied, the content_type argument cannot be None.

timeout:

This can be either an int or a float, indicating in how many seconds the request should timeout with an internally generated 408 response. If this is None, the internal 408 is only triggered by the internal PJSIP transaction timeout. Note that, even if the timeout is specified, the PJSIP timeout is also still valid.

```
end(self, timeout=None)
```

This will end the subscription by sending a SUBSCRIBE request with an Expires value of 0. If this object is no longer subscribed, this method will return without performing any action. This method cannot be called when the object is in the NULL state. The timeout argument has the same meaning as it does for the subscribe() method.

attributes

```
state
```

Indicates which state the internal state machine is in. See the previous section for a list of states the state machine can be in.

```
from_header
```

The FrozenFromHeader to be put in the From header of the SUBSCRIBE requests. This attribute is set on object instantiation and is read-only.

```
to_header
```

The FrozenToHeader that is the target for the subscription. This attribute is set on object instantiation and is read-only.

```
contact_header
```

The FrozenContactHeader to be put in the Contact header of the SUBSCRIBE requests. This attribute is set on object instantiation and is read-only.

```
event
```

The event package for which the subscription is as a string. This attribute is set on object instantiation and is read-only.

```
route_header
```

The outbound proxy to use in the form of a FrozenRouteHeader object. This attribute is set on object instantiation and is read-only.

```
credentials
```

The SIP credentials needed to authenticate at the SIP proxy in the form of a FrozenCredentials object. If none was specified when creating the Subscription object, this attribute is None. This attribute is set on object instantiation and is read-only.

```
refresh
```

The refresh interval in seconds that was requested on object instantiation as an int. This attribute is set on object instantiation and is read-only.

```
extra_headers
```

This contains the frozenlist of extra headers that was last passed to a successful call to the subscribe() method. If the argument was not provided, this attribute is an empty list. This attribute is read-only.

```
content_type
```

This attribute contains the Content-Type of the body that was last passed to a successful call to the subscribe() method. If the argument was not provided, this attribute is None.

```
body
```

This attribute contains the body string argument that was last passed to a successful call to the subscribe() method. If the argument was not provided, this attribute is None.

notifications

```
SIPSubscriptionChangedState
```

This notification will be sent every time the internal state machine of a Subscription object changes state.

timestamp:

A datetime.datetime object indicating when the notification was sent.

prev_state:

The previous state that the sate machine was in.

state:

The new state the state machine moved into.

```
SIPSubscriptionWillStart
```

This notification will be emitted when the initial SUBSCRIBE request is sent.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPSubscriptionDidStart
```

This notification will be sent when the initial SUBSCRIBE was accepted by the presence agent.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPSubscriptionGotNotify
```

This notification will be sent when a NOTIFY is received that corresponds to a particular Subscription object. Note that this notification could be sent when a NOTIFY without a body is received.

timestamp:

A datetime.datetime object indicating when the notification was sent.

method:

The method of the SIP request as a string. This will always be NOTIFY.

request_uri:

The request URI of the NOTIFY request as a SIPURI object.

headers:

The headers of the NOTIFY request as a dict. Each SIP header is represented in its parsed for as long as PJSIP supports it. The format of the parsed value depends on the header.

body:

The body of the NOTIFY request as a string, or None if no body was included. The content type of the body can be learned from the Content-Type header in the headers data attribute.

```
SIPSubscriptionDidFail
```

This notification will be sent whenever there is a failure, such as a rejected initial or refreshing SUBSCRIBE. After this notification the Subscription object is in the TERMINATED state and can no longer be used.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

An integer SIP code from the fatal response that was received from the remote party of generated by PJSIP. If the error is internal to the SIP core, this attribute will have a value of 0.

reason:

An text string describing the failure that occurred.

```
SIPSubscriptionDidEnd
```

This notification will be sent when the subscription ends normally, i.e. the end() method was called and the remote party sent a positive response. After this notification the Subscription object is in the TERMINATED state and can no longer be used.

timestamp:

A datetime.datetime object indicating when the notification was sent.

IncomingSubscription

Subscription and notifications for SIP events is an Internet standard published at RFC 3856.

This SIP primitive class is the mirror companion to the Subscription class and allows the application to take on the server role in a SUBSCRIBE session. This means that it can accept a SUBSCRIBE request and subsequent refreshing SUBSCRIBEs and sent NOTIFY requests containing event state.

In order to be able to receive SUBSCRIBE requests for a particular event package, the application needs to add the name of this event package to the incoming_events set attribute on the Engine, either at startup or at a later time, using the add_incoming_event() method. This event needs to be known by the Engine, i.e. it should already be present in the events dictionary attribute. Once the event package name is in the incoming_events set attribute, any incoming SUBSCRIBE request containing this name in the Event header causes the creation of a IncomingSubscribe object. This will be indicated to the application through a SIPIncomingSubscriptionGotSubscribe notification. It is then up to the application to check if the SUBSCRIBE request was valid, e.g. if it was actually directed at the correct SIP URI, and respond to it in a timely fashion.

The application can either reject the subscription by calling the reject() method or accept it through the accept() method, optionally first accepting it in the pending state by calling the accept_pending() method. After the IncomingSubscription is accepted, the application can trigger sending a NOTIFY request with a body reflecting the event state through the push_content() method. Whenever a refreshing SUBSCRIBE is received, the latest contents to be set are sent in the resulting NOTIFY request. The subscription can be ended by using the end() method.

methods

```
__init__(self)
```

An application should never create an IncomingSubscription object itself. Doing this will create a useless object in the None state.

```
reject(self, code)
```

Will reply to the initial SUBSCRIBE with a negative response. This method can only be called in the "incoming" state and sets the subscription to the "terminated" state.

code:

> The SIP response code to use. This should be a negative response, i.e. in the 3xx, 4xx, 5xx or 6xx range.

```
accept_pending(self)
```

Accept the initial incoming SUBSCRIBE, but put the subscription in the "pending" state and reply with a 202, followed by a NOTIFY request indicating the state. The

application can later decide to fully accept the subscription or terminate it. This method can only be called in the "incoming" state.

```
accept(self, content_type=None, content=None)
```

Accept the initial incoming SUBSCRIBE and respond to it with a 200 OK, or fully accept an IncomingSubscription that is in the "pending" state. In either case, a NOTIFY will be sent to update the state to "active", optionally with the content specified in the arguments. This method can only be called in the "incoming" or "pending" state.

content_type:

The Content-Type of the content to be set supplied as a string containing a "/" character. Note that if this argument is set, the content argument should also be set.

content:

The body of the NOTIFY to send when accepting the subscription, as a string. Note that if this argument is set, the content_type argument should also be set.

```
push_content(self, content_type, content)
```

Sets or updates the body of the NOTIFYs to be sent within this subscription and causes a NOTIFY to be sent to the subscriber. This method can only be called in the "active" state.

content_type:

The Content-Type of the content to be set supplied as a string containing a "/" character.

content:

The body of the NOTIFY as a string.

attributes

```
state
```

Indicates which state the incoming subscription session is in. This can be one of None, "incoming", "pending", "active" or "terminated". This attribute is read-only.

```
event
```

The name of the event package that this IncomingSubscription relates to. This attribute is a read-only string.

```
content_type
```

The Content-Type of the last set content in this subscription session. Inititally this will be None. This attribute is a read-only string.

```
content
```

The last set content in this subscription session as a read-only string.

notifications

```
SIPIncomingSubscriptionChangedState
```

This notification will be sent every time the an IncomingSubscription object changes state. This notification is mostly indicative, an application should not let its logic depend on it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

prev_state:

The previous state that the subscription was in.

state:

The new state the subscription has.

```
SIPIncomingSubscriptionGotSubscribe
```

This notification will be sent when a new IncomingSubscription is created as result of an incoming SUBSCRIBE request. The application should listen for this notification, retain a reference to the object that sent it and either accept or reject it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

method:

The method of the SIP request as a string, which will be SUBSCRIBE.

request_uri:

The request URI of the SUBSCRIBE request as a FrozenSIPURI object.

headers:

The headers of the SUBSCRIBE request as a dict, the values of which are the relevant header objects.

body:

The body of the SUBSCRIBE request as a string, or None if no body was included.

```
SIPIncomingSubscriptionGotRefreshingSubscribe
```

This notification indicates that a refreshing SUBSCRIBE request was received from the subscriber. It is purely informative, no action needs to be taken by the application as a result of it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

method:

The method of the SIP request as a string, which will be SUBSCRIBE.

request_uri:

The request URI of the SUBSCRIBE request as a FrozenSIPURI object.

headers:

The headers of the SUBSCRIBE request as a dict, the values of which are the relevant header objects.

body:

The body of the SUBSCRIBE request as a string, or None if no body was included.

`SIPIncomingSubscriptionGotUnsubscribe`

This notification indicates that a SUBSCRIBE request with an Expires header of 0 was received from the subscriber, effectively requesting to unsubscribe. It is purely informative, no action needs to be taken by the application as a result of it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

method:

The method of the SIP request as a string, which will be SUBSCRIBE.

request_uri:

The request URI of the SUBSCRIBE request as a FrozenSIPURI object.

headers:

The headers of the SUBSCRIBE request as a dict, the values of which are the relevant header objects.

body:

The body of the SUBSCRIBE request or response as a string, or None if no body was included.

`SIPIncomingSubscriptionAnsweredSubscribe`

This notification is sent whenever a response to a SUBSCRIBE request is sent by the object, including an unsubscribing SUBSCRIBE. It is purely informative, no action needs to be taken by the application as a result of it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The code of the SIP response as an int.

reason:

The reason text of the SIP response as an int.

headers:

The headers of the response as a dict, the values of which are the relevant header objects.

body:

This will be None.

```
SIPIncomingSubscriptionSentNotify
```

This notification indicates that a NOTIFY request was sent to the subscriber. It is purely informative, no action needs to be taken by the application as a result of it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

method:

The method of the SIP request as a string, which will be NOTIFY.

request_uri:

The request URI of the NOTIFY request as a FrozenSIPURI object.

headers:

The headers of the NOTIFY request as a dict, the values of which are the relevant header objects.

body:

The body of the NOTIFY request or response as a string, or None if no body was included.

```
SIPIncomingSubscriptionNotifyDidSucceed
```

This indicates that a positive final response was received from the subscriber in reply to a sent NOTIFY request. The notification is purely informative, no action needs to be taken by the application as a result of it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The code of the SIP response as an int.

reason:

The reason text of the SIP response as a string.

headers:

The headers of the response as a dict, the values of which are the relevant header objects.

body:

This will be None.

```
SIPIncomingSubscriptionNotifyDidFail
```

This indicates that a negative response was received from the subscriber in reply to a sent NOTIFY request or that the NOTIFY failed to send.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The code of the SIP response as an int. If this is 0, the notification was sent as a result of an internal error.

reason:

The reason text of the SIP response as a string or the internal error if the code attribute is 0.

headers: (if the notification was caused by a negative response)

The headers of the response as a dict, the values of which are the relevant header objects.

body: (if the notification was caused by a negative response)

This will be None.

```
SIPIncomingSubscriptionDidEnd
```

This notification is sent whenever the IncomingSubscription object reaches the "terminated" state and is no longer valid. After receiving this notification, the application should not longer try to use the object.

timestamp:

A datetime.datetime object indicating when the notification was sent.

Referral

Subscription and notifications for SIP events is an Internet standard published at RFC 3856. The REFER method, defined in RFC 3515 uses the subscription mechanism to tell SIP endpoints to refer to specific resources.

This SIP primitive class represents a referral requested by the client to a target URI. This means that the application should instance this class for each combination of target URI and resource it wishes the target to refer to. Whenever a NOTIFY is received, the application will receive the SIPReferralGotNotify notification.

Not creating an implicit subscription is supported as per RFC 4488

Internally a Referral object has a state machine, which reflects the state of the subscription. (The same as the Subscription since it uses the same event framework) It is a direct mirror of the state machine of the underlying pjsip_evsub object, whose possible states are at least NULL, SENT, ACCEPTED, PENDING, ACTIVE or TERMINATED. The last three states are directly copied from the contents of the Subscription-State header of the received NOTIFY request. Also, the state can be an arbitrary string if the contents of the Subscription-State header are not one of the three above. The state of the object is presented in the state attribute and changes of the state are indicated by means of the SIPReferralChangedState notification. This notification is only informational, an application using this object should take actions based on the other notifications sent by this object.

methods

```
__init__(self, request_uri, from_header, to_header,
refer_to_header, contact_header, route_header, credentials=None)
```

Creates a new Referral object which will be in the NULL state. The arguments for this method are documented in the attributes section above.

```
send_refer(self, create_subscription=1, extra_headers=list(),
timeout=None)
```

Calling this method triggers sending a REFER request to the presence agent. The arguments passed will be stored and used for subsequent refreshing SUBSCRIBE requests. The dialog may also be refreshed manually by using the refresh function. It may not be called when the object is in the TERMINATED state.

create_subscription:

Boolean flag indicating if an implicit subscription should be created.

extra_headers:

A list of additional headers to include in the REFER requests.

timeout:

This can be either an int or a float, indicating in how many seconds the request should timeout with an internally generated 408 response. If this is None, the

internal 408 is only triggered by the internal PJSIP transaction timeout. Note that, even if the timeout is specified, the PJSIP timeout is also still valid.

```
refresh(self, contact_header=None, extra_headers=list(),
timeout=None)
```

contact_header:

An optional ContactHeader object which will replace the local contact header and will be used from this moment on.

extra_headers:

A list of additional headers to include in the refreshing SUBSCRIBE request.

timeout:

The timeout argument has the same meaning as it does for the send_refer() method.

```
end(self, timeout=None)
```

This will end the referral subscription by sending a SUBSCRIBE request with an Expires value of 0. If this object is no longer subscribed, this method will return without performing any action. This method cannot be called when the object is in the NULL state. The timeout argument has the same meaning as it does for the send_refer() method.

attributes

```
state
```

Indicates which state the internal state machine is in. See the previous section for a list of states the state machine can be in.

```
from_header
```

The FrozenFromHeader to be put in the From header of the REFER and SUBSCRIBE requests. This attribute is set on object instantiation and is read-only.

```
to_header
```

The FrozenToHeader that is the target for the referral. This attribute is set on object instantiation and is read-only.

```
refer_to_header
```

The FrozenReferToHeader that is the target resource for the referral. This attribute is set on object instantiation and is read-only.

`local_contact_header`

The FrozenContactHeader to be put in the Contact header of the REFER and SUBSCRIBE requests. This attribute is set on object instantiation and is read-only.

`remote_contact_header`

The FrozenContactHeader received from the remote endpoint. This attribute is read-only.

`route_header`

The outbound proxy to use in the form of a FrozenRouteHeader object. This attribute is set on object instantiation and is read-only.

`credentials`

The SIP credentials needed to authenticate at the SIP proxy in the form of a FrozenCredentials object. If none was specified when creating the Referral object, this attribute is None. This attribute is set on object instantiation and is read-only.

`refresh`

The refresh interval in seconds that was requested on object instantiation as an int. This attribute is set when a NOTIFY request is received and is read-only.

`extra_headers`

This contains the frozenlist of extra headers that was last passed to a successful call to the subscribe() method. If the argument was not provided, this attribute is an empty list. This attribute is read-only.

`peer_address`

This read-only attribute contains the remote endpoint IP and port information. It can be accessed by accessing this object's ip and port attributes.

notifications

`SIPReferralChangedState`

This notification will be sent every time the internal state machine of a Referral object changes state.

timestamp:

A datetime.datetime object indicating when the notification was sent.

prev_state:

The previous state that the sate machine was in.

state:

The new state the state machine moved into.

```
SIPReferralWillStart
```

This notification will be emitted when the initial REFER request is sent.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPReferralDidStart
```

This notification will be sent when the initial REFER was accepted by the remote endpoint.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPReferralGotNotify
```

This notification will be sent when a NOTIFY is received that corresponds to a particular Referral object. Note that this notification could be sent when a NOTIFY without a body is received.

request_uri:

The request URI of the NOTIFY request as a SIPURI object.

from_header:

The From header of the NOTIFY request as a FrozenFromHeader object.

to_header:

The To header of the NOTIFY request as a FrozenToHeader object.

content_type:

The Content-Type header value of the NOTIFY request as a ContentType object.

event:

The Event header value of the NOTIFY request as a string object.

headers:

The headers of the NOTIFY request as a dict. Each SIP header is represented in its parsed for as long as PJSIP supports it. The format of the parsed value depends on the header.

body:

The body of the NOTIFY request as a string, or None if no body was included.

timestamp:

A datetime.datetime object indicating when the notification was sent.

```
SIPReferralDidFail
```

This notification will be sent whenever there is a failure, such as a rejected initial REFER or refreshing SUBSCRIBE. After this notification the Referral object is in the TERMINATED state and can no longer be used.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

An integer SIP code from the fatal response that was received from the remote party of generated by PJSIP. If the error is internal to the SIP core, this attribute will have a value of 0.

reason:

An text string describing the failure that occurred.

```
SIPReferralDidEnd
```

This notification will be sent when the subscription ends normally, i.e. the end() method was called and the remote party sent a positive response. It will also be sent when the remote endpoint sends a NOTIFY request with a noresource reason in the Subscription-State header. After this notification the Referral object is in the TERMINATED state and can no longer be used.

timestamp:

A datetime.datetime object indicating when the notification was sent.

IncomingReferral

Subscription and notifications for SIP events is an Internet standard published at RFC 3856. The REFER method, defined in RFC 3515 uses the subscription mechanism to to tell SIP endpoints to refer to specific resources.

This SIP primitive class is the mirror companion to the Referral class and allows the application to take on the server role in a REFER dialog. This means that it can accept a REFER request and subsequent refreshing SUBSCRIBEs and sent NOTIFY requests containing event state.

Any incoming REFER request causes the creation of a IncomingReferral object. This will be indicated to the application through a SIPIncomingReferralGotRefer notification. It is then up to the application to check if the REFER request was valid, e.g. if it was actually directed at the correct SIP URI, and respond to it in a timely fashion.

The application can either reject the referral by calling the reject() method or accept it through the accept() method. After the IncomingReferral is accepted, the application can trigger sending a NOTIFY request with a body reflecting the event state through the send_notify() method. Whenever a refreshing SUBSCRIBE is received, the latest contents to be set are sent in the resulting NOTIFY request. The subscription can be ended by using the end() method.

methods

```
__init__(self)
```

An application should never create an IncomingSubscription object itself. Doing this will create a useless object in the None state.

```
reject(self, code)
```

Will reply to the initial REFER with a negative response. This method can only be called in the "incoming" state and sets the referral to the "terminated" state.

code:

The SIP response code to use. This should be a negative response, i.e. in the 3xx, 4xx, 5xx or 6xx range.

```
accept(self, code=202, duration=180)
```

Accept the initial incoming REFER and respond to it with a 202 Accepted. A NOTIFY will be sent to update the state to "active", with a payload indicating the referral is in 100 Trying state. This method can only be called in the "incoming" state.

code:

The code to be used for the initial reply.

duration:

The desired duration for the implicit subscription. Unlike SUBSCRIBE initiated dialogs, in a referral the receiving party is the one choosing the expiration time.

```
send_notify(self, code, status=None)
```

Sets or updates the body of the NOTIFYs to be sent within this referral and causes a NOTIFY to be sent to the subscriber. This method can only be called in the "active" state.

code:

The response code to be used to generate the sipfrag payload.

status:

Optional status reason to be used to build the sipfrag payload. If none was specified the standard reason string will be used.

attributes

```
state
```

Indicates which state the incoming referral dialog is in. This can be one of None, "incoming", "pending", "active" or "terminated". This attribute is read-only.

```
local_contact_header
```

The FrozenContactHeader to be put in the Contact header of the REFER and SUBSCRIBE responses and NOTIFY requests. This attribute is set on object instantiation and is read-only.

```
remote_contact_header
```

The FrozenContactHeader received from the remote endpoint. This attribute is read-only.

```
peer_address
```

This read-only attribute contains the remote endpoint IP and port information. It can be accessed by accessing this object's ip and port attributes.

notifications

```
SIPIncomingReferralChangedState
```

This notification will be sent every time the an IncomingReferral object changes state. This notification is mostly indicative, an application should not let its logic depend on it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

prev_state:

The previous state that the subscription was in.

state:

The new state the subscription has.

```
SIPIncomingReferralGotRefer
```

This notification will be sent when a new IncomingReferral is created as result of an incoming REFER request. The application should listen for this notification, retain a reference to the object that sent it and either accept or reject it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

method:

The method of the SIP request as a string, which will be REFER.

request_uri:

The request URI of the REFER request as a FrozenSIPURI object.

refer_to:

The Refer-To header as a FrozenReferToHeader object.

headers:

The headers of the REFER request as a dict, the values of which are the relevant header objects.

body:

The body of the REFER request as a string, or None if no body was included.

```
SIPIncomingReferralGotRefreshingSubscribe
```

This notification indicates that a refreshing SUBSCRIBE request was received from the subscriber. It is purely informative, no action needs to be taken by the application as a result of it.

```
SIPIncomingReferralGotUnsubscribe
```

This notification indicates that a SUBSCRIBE request with an Expires header of 0 was received from the subscriber, effectively requesting to unsubscribe. It is purely informative, no action needs to be taken by the application as a result of it.

`SIPIncomingReferralAnsweredRefer`

This notification is sent whenever a response to a REFER request is sent by the object. It is purely informative, no action needs to be taken by the application as a result of it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The code of the SIP response as an int.

reason:

The reason text of the SIP response as an int.

headers:

The headers of the response as a dict, the values of which are the relevant header objects.

body:

This will be None.

`SIPIncomingReferralSentNotify`

This notification indicates that a NOTIFY request was sent to the subscriber. It is purely informative, no action needs to be taken by the application as a result of it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

method:

The method of the SIP request as a string, which will be NOTIFY.

request_uri:

The request URI of the NOTIFY request as a FrozenSIPURI object.

headers:

The headers of the NOTIFY request as a dict, the values of which are the relevant header objects.

body:

The body of the NOTIFY request or response as a string, or None if no body was included.

`SIPIncomingReferralNotifyDidSucceed`

This indicates that a positive final response was received from the subscriber in reply to a sent NOTIFY request. The notification is purely informative, no action needs to be taken by the application as a result of it.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The code of the SIP response as an int.

reason:

The reason text of the SIP response as a string.

headers:

The headers of the response as a dict, the values of which are the relevant header objects.

body:

This will be None.

```
SIPIncomingReferralNotifyDidFail
```

This indicates that a negative response was received from the subscriber in reply to a sent NOTIFY request or that the NOTIFY failed to send.

timestamp:

A datetime.datetime object indicating when the notification was sent.

code:

The code of the SIP response as an int. If this is 0, the notification was sent as a result of an internal error.

reason:

The reason text of the SIP response as a string or the internal error if the code attribute is 0.

headers: (if the notification was caused by a negative response)

The headers of the response as a dict, the values of which are the relevant header objects.

body: (if the notification was caused by a negative response)

This will be None.

```
SIPIncomingReferralDidEnd
```

This notification is sent whenever the IncomingReferral object reaches the "terminated" state and is no longer valid. After receiving this notification, the application should not longer try to use the object.

timestamp:

A datetime.datetime object indicating when the notification was sent.

AudioMixer

An AudioMixer manages two audio devices, on for input and one for output, and is able to route audio data for a number of sources. It wraps a PJSIP conference bridge, the concept of which is explained in the PJSIP documentation. Any component in the core that either produces or consumes sound, i.e. AudioTransport, ToneGenerator, WaveFile and RecordingWaveFile objects, has a ConferenceBridge associated with it and a corresponding slot on that conference bridge. The sound devices, both input and output, together always occupy slot 0. It is up to the application to setup the desired routing between these components. Note that the middleware provides an abstracted API which hides the complexity of using the low-level PJSIP concepts. This is mainly provided in the {{{sipsimple.audio}}} module, but also consists of other audio-enabled objects (such as the AudioStream).

methods

```
__init__(self, input_device, output_device, sample_rate,
ec_tail_length=200, slot_count=254)
```

Creates a new ConferenceBridge object.

input_device:

The name of the audio input device to use as a string, or None if no input device is to be used. A list of known input devices can be queried through the Engine.input_devices attribute. If "system_default" is used, PJSIP will select the system default output device, or None if no audio input device is present. The device that was selected can be queried afterwards through the input_device attribute.

output_device:

The name of the audio output device to use as a string, or None if no output device is to be used. A list of known output devices can be queried through the Engine.output_devices attribute. If "system_default" is used, PJSIP will select the system default output device, or None if no audio output device is present. The device that was selected can be queried afterwards through the output_device attribute.

sample_rate:

The sample rate on which the conference bridge and sound devices should operate in Hz. This must be a multiple of 50.

ec_tail_length:

The echo cancellation tail length in ms. If this value is 0, no echo cancellation is used.

slot_count:

The number of slots to allocate on the conference bridge. Not that this includes the slot that is occupied by the sound devices.

```
set_sound_devices(self, input_device, output_device,
ec_tail_length)
```

Change the sound devices used (including echo cancellation) by the conference bridge. The meaning of the parameters is that same as for __init__().

```
connect_slots(self, src_slot, dst_slot)
```

Connect two slots on the conference bridge, making audio flow from src_slot to dst_slot.

```
disconnect_slots(self, src_slot, dst_slot)
```

Break the connection between the specified slots. Note that when an audio object is stopped or destroyed, its connections on the conference bridge will automatically be removed.

attributes

```
input_device
```

A string specifying the audio input device that is currently being used. If there is no input device, this attribute will be None. This attribute is read-only, but may be updated by calling the set_sound_devices() method.

```
output_device
```

A string specifying the audio output device that is currently being used. If there is no output device, this attribute will be None. This attribute is read-only, but may be updated by calling the set_sound_devices() method.

```
sample_rate
```

The sample rate in Hz that the conference bridge is currently operating on. This read-only attribute is passed when the object is created.

```
ec_tail_length
```

The current echo cancellation tail length in ms. If this value is 0, no echo cancellation is used. This attribute is read-only, but may be updated by calling the set_sound_devices() method.

```
slot_count
```

The total number of slots that is available on the conference bridge This read-only attribute is passed when the object is created.

```
used_slot_count
```

A read-only attribute indicating the number of slots that are used by objects.

```
input_volume
```

This writable property indicates the % of volume that is read from the audio input device. By default this value is 100.

```
output_volume
```

This writable property indicates the % of volume that is sent to the audio output device. By default this value is 100.

```
muted
```

This writable boolean property indicates if the input audio device is muted.

```
connected_slots
```

A read-only list of tupples indicating which slot is connected to which. Connections are directional.

MixerPort

This a simple object which simply copies all the audio data it gets as input to it output. It's main purpose is that of facilitating the creation the of the middleware AudioBridge object.

methods

```
__init__(self, mixer)
```

Create a new MixerPort which is associated with the specified AudioMixer.

```
start(self)
```

Activate the mixer port. This will reserve a slot on the AudioMixer and allow it to be connected to other slots.

```
stop(self)
```

Deactivate the mixer port. This will release the slot previously allocated on the AudioMixer and all connections which it had will be discarded.

attributes

```
mixer
```

The AudioMixer this MixerPort is associated with This attribute is read-only.

```
is_active
```

Whether or not this MixerPort has a slot associated in its AudioMixer. This attribute is read-only.

```
slot
```

The slot this MixerPort has reserved on AudioMixer or None if it is not active. This attribute is read-only.

WaveFile

This is a simple object that allows playing back of a .wav file over the PJSIP conference bridge. Only 16-bit PCM and A-law/U-law formats are supported. Its main purpose is the playback of ringtones or previously recorded conversations.

This object is associated with a AudioMixer instance and, once the start() method is called, connects to it and sends the sound to all its connections. Note that the slot of the WaveFile object will not start playing until it is connected to another slot on the AudioMixer. If the stop() method is called or the end of the .wav file is reached, a WaveFileDidFinishPlaying notification is sent by the object. After this the start() method may be called again in order to re-use the object.

It is the application's responsibility to keep a reference to the WaveFile object for the duration of playback. If the reference count of the object reaches 0, playback will be stopped automatically. In this case no notification will be sent.

methods

```
__init__(self, mixer, filename)
```

Creates a new WaveFile object.

mixer:

The AudioMixer object that this object is to be connected to.

filename:

The name of the .wav file to be played back as a string. This should include the full path to the file.

```
start(self)
```

Start playback of the .wav file.

```
stop(self)
```

Stop playback of the file.

attributes

```
mixer
```

The AudioMixer this object is associated with. This attribute is read-only.

```
filename
```

The name of the .wav file, as specified when the object was created. This attribute is read-only.

```
slot
```

A read-only property indicating the slot number at which this object is attached to the associated AudioMixer. If the WaveFile is not active, this attribute will be None.

```
volume
```

A writable property indicating the % of volume at which this object contributes audio to the AudioMixer. By default this is set to 100.

```
is_active
```

A boolean read-only property that indicates if the file is currently being played.

notifications

```
WaveFileDidFinishPlaying
```

This notification will be sent whenever playing of the .wav has stopped. After sending this event, the playback may be re-started.

timestamp:

A datetime.datetime object indicating when the notification was sent.

RecordingWaveFile

This is a simple object that allows recording audio to a PCM .wav file.

This object is associated with a AudioMixer instance and, once the start() method is called, crecords sound from its connections. Note that the RecordingWaveFile will not record anything if it does not have any connections. Recording to the file can be stopped either by calling the stop() method or by removing all references to the object. Once the stop() method has been called, the start() method may not be called again. It is the application's responsibility to keep a reference to the RecordingWaveFile object for the duration of the recording, it will be stopped automatically when the reference count reaches 0.

methods

```
__init__(self, mixer, filename)
```

Creates a new RecordingWaveFile object.

mixer:

The AudioMixer object that this object is to be associated with.

filename:

The name of the .wav file to record to as a string. This should include the full path to the file.

```
start(self)
```

Start recording the sound to the .wav file.

```
stop(self)
```

Stop recording to the file.

attributes

```
mixer
```

The AudioMixer this object is associated with. This attribute is read-only.

```
filename
```

The name of the .wav file, as specified when the object was created. This attribute is read-only.

```
slot
```

A read-only property indicating the slot number at which this object is attached to the associated AudioMixer. If the RecordingWaveFile is not active, this attribute will be None.

```
is_active
```

A boolean read-only attribute that indicates if the file is currently being written to.

ToneGenerator

A ToneGenerator can generate a series of dual frequency tones and has a shortcut method for generating valid DTMF tones. Each instance of this object is associated with a AudioMixer object, which it will connect to once started. The tones will be sent to the slots on the AudioMixer its slot is connected to. Once started, a ToneGenerator can be stopped by calling the stop() method and is automatically destroyed when the reference count reaches 0.

> Note: this object has threading issues when the application uses multiple AudioMixers. It should not be used.

methods

```
__init__(self, mixer)
```

Creates a new ToneGenerator object.

mixer:

> The AudioMixer object that this object is to be connected to.

start(self)

> Start the tone generator and connect it to its associated AudioMixer.

stop(self)

> Stop the tone generator and remove it from the conference bridge.

play_tones(self, tones)

> Play a sequence of single or dual frequency tones over the audio device.

tones:

> This should be a list of 3-item tuples, in the form of [(<freq1>, <freq2>, <duration>), ...], with Hz as unit for the frequencies and ms as unit for the duration. If freq2 is 0, this indicates that only freq1 should be used for the tone. If freq1 is 0, this indicates a pause when no tone should be played for the set duration.

```
play_dtmf(self, digit)
```

Play a single DTMF digit.

digit:

> A string of length 1 containing a valid DTMF digit, i.e. 0 through 9, #, * or A through D.

attributes

```
mixer
```

The AudioMixer this object is associated with. This attribute is read-only.

```
slot
```

A read-only property indicating the slot number at which this object is attached to the associated AudioMixer. If the ToneGenerator has not been started, this attribute will be None.

```
volume
```

A writable property indicating the % of volume at which this object contributes audio. By default this is set to 100.

```
is_active
```

A boolean read-only property that indicates if the object has been started.

```
is_busy
```

A boolean read-only property indicating if the ToneGenerator is busy playing tones.

notifications

```
ToneGeneratorDidFinishPlaying
```

This notification will be sent whenever playing of the queued tones has finished.

timestamp:

A datetime.datetime object indicating when the notification was sent.

MSRP API

Message Session Relay Protocol (MSRP) is a protocol for transmitting a series of related Instant Messages in the context of a session. Message sessions are treated like any other media stream when set up via a rendezvous or session creation protocol such as the Session Initiation Protocol (SIP).

- MSRP sessions are defined in RFC 4975
- MSRP relay extension used for NAT traversal of instant messaging and file transfer sessions is defined in RFC 4976

The MSRP protocol stack is implemented by msrplib Python package.

msrplib is based upon twisted and eventlet and provides a set of classes for establishing and managing MSRP connections.

The library consists of the following modules:

msrplib.transport

Defines MSRPTransport class, which provides low level control over MSRP connections.

msrplib.connect

Defines means to establish a connection, bind it, and provide an initialized MSRPTransport instance.

msrplib.session

Defines MSRPSession class, which provides high level control over a MSRP connection.

msrplib.protocol

Provides representation and parsing of MSRP entities - chunks and MSRP URIs.

URI

This class is implemented in the msrplib.protocol module and is used to represent an MSRP URI.

methods

```
__init__(self, host=None, use_tls=False, user=None, port=None,
session_id=None, parameters=None, credentials=None)
```

Constructs a new URI. All the arguments specified here are also attributes on the object. For more information on these attributes, see RFC4975.

host:

The hostname or IP address which forms the URI.

use_tls:

Whether this identifies an msrps or msrp URI.

user:

The user part of the URI.

port:

The port in the URI.

session_id:

The session identifier part of the URI.

parameters:

A dict containing the parameters which will be appended to the URI.

credentials:

A gnutls.interfaces.twisted.X509Credentials object which will be used if this identifies a TLS URI to authenticate to the other endpoint.

MSRPRelaySettings

This class is implemented in the msrplib.connect module and is used to specify the MSRP relay which will be used when connecting via a relay (using the ConnectorRelay or AcceptorRelay classes).

methods

```
__init__(self, domain, username, password, host=None,
use_tls=False, port=None, credentials=None)
```

Constructs a new URI. All the arguments specified here are also attributes on the object. For more information on these attributes, see RFC4975.

domain:

The DNS domain in which to search for a MSRP relay using SRV queries.

username:

The username which will be used to authenticate to the relay.

password:

The password which will be used to authenticate to the relay.

host:

The hostname or IP address of the MSRP relay.

use_tls:

Whether this identifies an msrps or msrp URI.

port:

The port in the URI.

credentials:

A gnutls.interfaces.twisted.X509Credentials object which will be used to authenticate to the relay if TLS is used.

ConnectorDirect

This class is implemented in the msrplib.connect module and is used for outbound MSRP connections without a relay.

methods

```
__init__(self, loger=None, use_sessmatch=False)
```

Constructs a new ConnectorDirect.

logger:

The logger which will be used for this MSRP connection. See the Logging section for more information.

use_sessmatch:

Indicates if the connector should use the session matching mechanism defined by the http://tools.ietf.org/html/draft-ietf-simple-msrp-sessmatch-10

```
prepare(self, local_uri=None)
```

This method returns a full local path - list of protocol.URI instances, suitable to be put in SDP 'a:path' attribute.

local_uri:

This attribute will be used to construct the local path, but other than that it is not used anywhere else in case of the ConnectorDirect. If not provided, one will be automatically generated

```
complete(self, full_remote_path)
```

This method establishes the connection and binds it (sends an empty chunk to verify each other's From-Path and To-Path). It returns transport.MSRPTransport instance, ready to read and send chunks.

full_remote_path:

A list of protocol.URI instances, obtained by parsing 'a:path' put in SDP by the remote party.

```
cleanup(self)
```

This method cleans up after prepare(); it should be called if complete() will not be called for whatever reason.

AcceptorDirect

This class is implemented in the msrplib.connect module and is used for inbound MSRP connections without using a relay.

methods

```
__init__(self, loger=None, use_sessmatch=False)
```

Constructs a new AcceptorDirect.

logger:

The logger which will be used for this MSRP connection. See the Logging section for more information.

use_sessmatch:

Indicates if the connector should use the session matching mechanism defined by the http://tools.ietf.org/html/draft-ietf-simple-msrp-sessmatch-10

```
prepare(self, local_uri=None)
```

This method starts listening on a socket identified by the parameters in the local_uri argument. It returns a full local path - list of protocol.URI instances, suitable to be put in SDP 'a:path' attribute.

local_uri:

This attribute will be used to construct the local path and to listen for incomming connections. If not provided, one will be automatically generated. Note that the object may be changed in place: for example, if the port specified is 0, a random one will be selected and the object will be updated accordingly.

```
complete(self, full_remote_path)
```

This method waits for an incoming connection and a chunk sent by the other party. It returns transport.MSRPTransport instance, ready to read and send chunks.

full_remote_path:

A list of protocol.URI instances, obtained by parsing 'a:path' put in SDP by the remote party. This is checked agains the From-Path header in the binding chunk.

```
cleanup(self)
```

This method cleans up after prepare(); it should be called if complete() will not be called for whatever reason.

RelayConnection

This class is implemented in the msrplib.connect module and is used for inbound and outbound MSRP connections using a relay.

methods

```
__init__(self, relay, mode, loger=None, use_sessmatch=False)
```

Constructs a new RelayConnection.

relay:

An instance of MSRPRelaySettings which identifies the relay which is to be used.

mode:

A string indicating if this connection should be active, an empty SEND should be sent when complete is called, or passive, where it will wait for one.

logger:

The logger which will be used for this MSRP connection. See the Logging section for more information.

use_sessmatch:

Indicates if the connector should use the session matching mechanism defined by the http://tools.ietf.org/html/draft-ietf-simple-msrp-sessmatch-10

```
prepare(self, local_uri=None)
```

This method returns a full local path - list of protocol.URI instances, suitable to be put in SDP 'a:path' attribute.

local_uri:

This attribute will be used to construct the local path, but other than that it is not used anywhere else in case of the ConnectorRelay. If not provided, one will be automatically generated

```
complete(self, full_remote_path)
```

This method establishes the connection to the relay and binds it (sends an empty chunk or waits for one, depending on the mode, to verify each other's From-Path and To-Path). It returns transport.MSRPTransport instance, ready to read and send chunks.

full_remote_path:

A list of protocol.URI instances, obtained by parsing 'a:path' put in SDP by the remote party.

```
cleanup(self)
```

This method cleans up after prepare(); it should be called if complete() will not be called for whatever reason.

MSRPTransport

This class is implemented in the msrplib.transport module and provies low level access to the MSRP connection. This class should not be constructed directly, but rather its intances should be obtained by using the various connector/acceptor classes documented above.

methods

```
make_chunk(self, transaction_id=None, method='SEND', code=None,
comment=None, data='', contflag=None, start=1, end=None,
length=None, message_id=None)
```

Creates a new chunk (protocol.MSRPData instance), which is either an MSRP request or a response. Proper From-Path, To-Path, Byte-Range and Message-ID headers are added based on MSRPTransport's state and the parameters provided. Use data for payload, and start/end/length to generate the Byte-Range header.

transaction_id:

The transaction id which will be put in the chunk. If it is not provided, one will be randomly generated.

method:

The method of the new MSRP request, or None if a response is required.

code:

The code of the new MSRP response, or None if a request is required.

code:

The comment of the new MSRP response, or None if a request is required or a comment on the response is not.

data:

The payload of the new chunk, or an empty string if no payload is required.

contflag:

MSRP chunk's continuation flag ('$', '+' or '#'). Default is '$' for a full message, unless a partial SEND chunk required, in which case it should be set to '+'. If None is provided, either '$' or '+' will be used depending on whether this chunk seems to carry the last part of the message data.

start:

The first byte's index within the whole message payload this chunk will carry as its own payload.

end:

The last byte's index within the whole message payload this chunk will carry as its own payload. Note that this is an inclusive index. If this is not provided, it's computed based on the number of bytes in data and start.

length:

The total number of bytes of the whole message payload. If this is not provided, it is computed as if this chunk carries that last part of the message payload.

message_id:

The ID of the message this chunk is part of. If it is not provided, one will be randomly generated.

```
write_chunk(self, chunk, wait=True)
```

Writes the chunk provided to the underlying socket.

chunk:

The chunk which is to be written, an instance of MSRPData.

wait:

If True, waits for the operation to complete.

```
write_response(self, chunk, code, comment, wait=True)
```

Creates a response which is suitable as a reply to the specified chunk and writes it to the underlying socket. chunk: The chunk for which to create a response. code: The status code of the response which is to be created. comment: The comment of the response which is to be created. wait: If True, waits for the operation to complete.

```
read_chunk(self, size=None)
```

Waits for a new chunk and returns it. If there was an error, closes the connection and raises ChunkParseError.

In case of unintelligible input, loses the connection and returns None. When the connection is closed, raises the reason of the closure (e.g. ConnectionDone).

If the data already read exceeds size, stops reading the data and returns a "virtual" chunk, i.e. the one that does not actually correspond the the real MSRP chunk. Such chunks have Byte-Range header changed to match the number of bytes read and continuation that is '+'; they also possess segment attribute, an integer, starting with 1 and increasing with every new segment of the chunk.

Note, that size only hints when to interrupt the segment but does not affect how the data is read from socket. You may have segments bigger than size and it's legal to set size to zero (which would mean return a chunk as long as you get some data, regardless how small).

size:

The hint towards how much to read from the socket. If the data already read is larger, then all the data will be returned, even if it exceeds size bytes.

```
check_incoming_SEND_chunk(self, chunk)
```

Checks the 'To-Path' and 'From-Path' of the incoming SEND chunk. Returns None is the paths are valid for this connection. If an error is detected an MSRPError is created and returned.

MSRPData

This class is implemented in the msrplib.protocol module and represents an MSRP chunk, either a request or a response.

attributes

The following attributes are read-only.

content_type

The MIME type of the payload carried by this chunk, which is stored in the Content-Type header.

message_id

The ID of the message this chunk is part of, which is stored in the Message-ID header.

byte_range

A 3-tuple containing the start, end and length values (in this order) from the Byte-Range header.

status

The value of the Status header.

failure_report

The value of the Failure-Report header, whether it exists or it is implied; one of 'yes', 'no', 'partial'.

success_report

The value of the Success-Report header, whether it exists or it is implied; one of 'yes' or 'no'.

size

The size of the payload of this chunk, in bytes.

methods

```
__init__(self, transaction_id, method=None, code=None,
comment=None, headers=None, data='', contflag='$')
```

Initializes a new MSRPData instance. All the arguments are also available as attributes.

transaction_id:

The transaction identified of this chunk.

method:

The method of this chunk if it a request and None if it is a response.

code:

The status code of this chunk if it is a response and None if it is a request.

comment:

The comment of this chunk if it is a response and None if it is a request.

headers:

A dict containing the headers which are to be added to this chunk, or None if no headers are to be added.

data:

The payload of this chunk, or an empty string if no payload is to be added.

contflag:

The MSRP continuation flag of this chunk, one of '$', '#' or '+'.

```
copy(self)
```

Returns a new MSRPData instance with exactly the same attributes as this object.

```
add_header(self, header)
```

Add a MSRP header to this chunk.

header:

The header object which is to be added to this chunk.

```
verify_headers(self)
```

Verifies that the chunk contains a To-Path and a From-Path header and that all the headers are valid.

```
encode_start(self)
```

Returns a string containing the MSRP header of this chunk.

```
encode_end(self, continuation)
```

Returns a string containing the MSRP end line of this chunk: 7 commas followed by the transaction identifier and the continuation flag specified.

continuation:

The continuation flag which is to be used in the end line.

```
encode(self):
```

Returns a string containing the whole of this MSRP chunk.

OutgoingFile

This class is implemented in the msrplib.session module and represents a file which is to be sent via MSRP.

attributes

headers

A dict which maps header names to header objects. These headers will be sent in the MSRP message used to send the file.

methods

```
__init__(self, fileobj, size, content_type=None, position=0,
message_id=None)
```

Initializes a new OutgoingFile using the specified file object. All arguments are available as attributes, except for content_type which will be part of the headers attribute.

fileobj:

A file-like object which will be used for reading the data to be sent.

size:

The size of the whole file.

content_type:

The MIME type associated with the file's data. If provided, it will be added as a Content-Type header.

position:

The position within the file from which data will be sent. The file object must already be seeked to this position.

message_id:

The message ID which will be used for sending this file.

MSRPSession

This class is implemented in the msrplib.session module and provides a high level API for MSRP sessions.

methods

```
__init__(self, msrptransport, accept_types=['*'],
on_incoming_cb=None)
```

Initializes MSRPSession instance over the specified transport. The incoming chunks are reported through the on_incoming_cb callback.

msrptransport:

An instance of MSRPTransport over which this session will operate.

accept_types:

A list of MIME types which are acceptable over this session. If data is received with a Content-Type header which doesn't match this list, a negative response is sent back and the application does not get the received data. The special strings '*' and '<type>/*' can be used to match multiple MIME types.

on_incoming_cb:

A function which receives two arguments, both optional with default values of None: chunk and error. This will be called when a new chunk is received.

```
send_chunk(self, chunk, response_cb=None)
```

Sends the specified chunk and reports the result via the response_cb callback.

When response_cb argument is present, it will be used to report the transaction response to the caller. When a response is received or generated locally, response_cb is called with one argument. The function must do something quickly and must not block, because otherwise it would block the reader greenlet.

If no response was received after RESPONSE_TIMEOUT seconds,

1. 408 response is generated if Failure-Report was 'yes' or absent
2. 200 response is generated if Failure-Report was 'partial' or 'no'

Note that it's rather wasteful to provide response_cb argument other than None for chunks with Failure-Report='no' since it will always fire 30 seconds later with 200 result (unless the other party is broken and ignores Failure-Report header)

If sending is not possible, MSRPSessionError is raised.

chunk:

The chunk which is to be sent. It must be an instance of MSRPData which represents a request.

response_cb:

A function receiving a single argument which will be the response received for the sent chunk.

```
deliver_chunk(self, chunk, event=None)
```

Sends the specified chunk and waits for the transaction response (if Failure-Report header is not 'no'). Returns the transaction response if it's a success or raise MSRPTransactionError if it's not.

If chunk's Failure-Report is 'no', returns None immediately.

chunk:

The chunk which is to be sent. It must be an intance of MSRPData which represents a request.

event:

The eventlet.coros.event object which will be used to wait for a response. It's send method will be called when a response is received. If it is not provided, one will be constructed automatically and used.

```
send_file(self, outgoing_file)
```

Adds the specified file to the queue of files to be sent. The method returns immediately.

outgoing_file:

An instance of OutgoingFile which represents the file to be sent.

```
shutdown(self, sync=True)
```

Sends the messages already in queue then closes the connection.

MSRPServer

This class is implemented in the msrplib.connect module. MSRPServer solves the problem with AcceptorDirect: concurrent using of 2 or more AcceptorDirect instances on the same non-zero port is not possible. If you initialize() those instances, one after another, one will listen on the socket and another will get BindError.

MSRPServer avoids the problem by sharing the listening socket between multiple connections. It has a slightly different interface from AcceptorDirect, so it cannot be considered a drop-in replacement.

It manages the listening sockets and binds incoming requests.

methods

```
__init__(self, logger)
```

Constructs a new MSRPServer which will be using the specifed logger for its connections.

logger:

The default logger which will be used for this object's MSRP connections. See the Logging section for more information.

```
prepare(self, local_uri=None, logger=None)
```

Starts a listening port specified by local_uri if there isn't one on that port/interface already. Adds local_uri to the list of expected URIs, so that incoming connections featuring this URI won't be rejected. If logger is provided, it uses it for this connection instead of the one specified in __init___.

local_uri:

The URI which will be added to the list of expected URIs. Connections from this URI will be accepted.

logger:

The logger which will be used for connections from the specified URI. See the Logging section for more information.

```
complete(self, full_remote_path)
```

Waits until one of the incoming connections binds using the provided full_remote_path. Returns the connected and bound MSRPTransport instance. If no such binding was made within MSRPBindSessionTimeout.seconds, MSRPBindSessionTimeout is raised.

full_remote_path:

A list of protocol.URI instances, obtained by parsing 'a:path' put in SDP by the remote party.

```
cleanup(self, local_uri)
```

Removes local_uri from the list of expected URIs.

local_uri:

The URI which is to be removed from the list of expected URIs.

Headers

The MSRP headers are represented using objects from the msrplib.protocol module. All MSRP header object provide three properties:

name: The name of the header, as it appears in MSRP chunks. encode: The MSRP encoded header value, as it appears in MSRP chunks. decode: The high-level object representing the header value.

All headers can be constructed by passing either the encoded or decoded objects to __init__. The following headers are provided:

ToPathHeader

The name of the header is 'To-Path' and the decoded value is a deque of URI objects.

FromPathHeader

The name of the header is 'From-Path' and the decoded value is a deque of URI objects.

MessageIDHeader

The name of the header is 'Message-ID' and the decoded value is the string containing the message ID.

SuccessReportHeader

The name of the header is 'Success-Report' and the decoded value is one of 'yes' or 'no'.

FailureReportHeader

The name of the header is 'Failure-Report' and the decoded value is one of 'yes', 'partial' or 'no'.

ByteRangeHeader

The name of the header is 'Byte-Range' and the decoded value is a 3-tuple containing the start, end and length values.

attributes

fro

The start value of the header: the index within the whole message payload where a chunk's payload starts.

end

The end value of the header: the index within the whole message payload where a chunk's payload ends. Note that this index is inclusive.

length

The total size of the message payload.

StatusHeader

The name of the header is 'Status' and the decoded value is a 2-tuple containing the status code and comment.

attributes

code

The code component of the header.

comment

The comment component of the header.

ExpiresHeader

The name of the header is 'Expires' and the decoded value is an integer.

MinExpiresHeader

The name of the header is 'Min-Expires' and the decoded value is an integer.

MaxExpiresHeader

The name of the header is 'Max-Expires' and the decoded value is an integer.

UsePathHeader

The name of the header is 'Use-Path' and the decoded value is a deque of URI objects.

WWWAuthenticateHeader

The name of the header is 'WWW-Authenticate' and the decoded value is a dictionary of the parameters within the header.

AuthorizationHeader

The name of the header is 'Authorization' and the decoded value is a dictionary of the parameters within the header.

AuthenticationInfoHeader

The name of the header is 'Authentication-Info' and the decoded value is a dictionary of the parameters within the header.

ContentTypeHeader

The name of the header is 'Content-Type' and the decoded value is the string representing the MIME type.

ContentIDHeader

The name of the header is 'Content-ID' and the decoded value is the string with the value of the header.

ContentDescriptionHeader

The name of the header is 'Content-Description' and the decoded value is the string with the value of the header.

ContentDispositionHeader

The name of the header is 'Content-Disposition' and the decoded value is a list with two elements: the disposition and a dict with the parameters.

Logging

Logging is done throughout the library using objects defined by the application, called loggers. If logging is not desired, the application.python.Null object of python-application can be used. These loggers must define the following methods:

methods

```
received_new_chunk(data, transport, chunk)
```

This method is called when the start of a new chunk is received.

data:

The data which came along with the start of the chunk.

transport:

The MSRPTransport instance on which the chunk was received.

chunk:

The actual chunk object.

```
received_chunk_data(data, transport, transaction_id)
```

This method is called when data is received as part of a chunk previously announced via received_new_chunk.

data:

The data received as part of the chunk.

transport:

The MSRPTransport instance on which the chunk was received.

transaction_id:

The transaction ID which identifies the chunk for which data was received.

```
received_chunk_end(data, transport, transaction_id)
```

This method is called when the last data of a chunk is received. The chunk was previously announced via received_new_chunk.

data:

The last data received as part of the chunk.

transport:

The MSRPTransport instance on which the chunk was received.

transaction_id:

The transaction ID which identifies the chunk which was ended.

```
sent_new_chunk(data, transport, chunk)
```

This method is called when the start of a new chunk is sent.

data:

The data which was sent along with the start of the chunk.

transport:

The MSRPTransport instance on which the chunk was sent.

chunk:

The actual chunk object.

```
sent_chunk_data(data, transport, transaction_id)
```

This method is called when data is sent as part of a chunk previously announced via sent_new_chunk.

data:

The data sent as part of the chunk.

transport:

The MSRPTransport instance on which the chunk was sent.

transaction_id:

The transaction ID which identifies the chunk for which data was sent.

```
sent_chunk_end(data, transport, transaction_id)
```

This method is called when the last data of a chunk is sent. The chunk was previously announced via sent_new_chunk.

data:

The last data sent as part of the chunk.

transport:

The MSRPTransport instance on which the chunk was sent.

transaction_id:

The transaction ID which identifies the chunk which was ended.

```
debug(message)
```

This method is called when a debug level message is sent by the library.

```
info(message)
```

This method is called when a info level message is sent by the library.

```
warn(message)
```

This method is called when a warning level message is sent by the library.

```
error(message)
```

This method is called when a error level message is sent by the library.

```
fatal(message)
```

This method is called when a fatal level message is sent by the library.

Examples

Creating an outbound connection

When creating an outbound connection, a relay is not usually used because NAT traversal is not a problem for the endpoing creating the connection. If one is nevertheless desired, a ConnectorRelay can be used instead:

```
from msrplib.connect import ConnectorDirect
from msrplib.session import MSRPSession

connector = ConnectorDirect()
full_local_path = connector.prepare()
try:
    # put full_local_path in SDP 'a:path' attribute of offer
    # get full_remote_path from remote's 'a:path: attribute of reply
    msrp_transport = connector.complete(full_remote_path)
except:
    connector.cleanup()
    raise

def handle_incoming(chunk=None, error=None):
    if error is not None:
       print 'Error: %s' % error
       session.shutdown()
    elif chunk is not None:
       print 'Got data: %s' % chunk.data

session = MSRPSession(msrp_transport, on_incoming_cb=handle_incoming)
session.send_chunk(msrp_transport.make_chunk(data='Hello world!'))
```

Waiting for an inbound connection

When creating an inbound connection, a relay must be used for NAT traversal. However, if one is not desired, an AceptorDirect can be used instead:

```
from msrplib.connect import RelayConnection, MSRPRelaySettings
from msrplib.session import MSRPSession

relay = MSRPRelaySettings(domain='example.org', username='user',
password='pass')
connector = RelayConnection(relay, 'passive')
full_local_path = connector.prepare()
try:
    # get full_remote_path from remote's 'a:path: attribute of offer
    # put full_local_path in SDP 'a:path' attribute of reply
    msrp_transport = connector.complete(full_remote_path)
except:
    connector.cleanup()
    raise

def handle_incoming(chunk=None, error=None):
    if error is not None:
       print 'Error: %s' % error
```

```
        session.shutdown()
    elif chunk is not None:
        print 'Got data: %s' % chunk.data

session = MSRPSession(msrp_transport, on_incoming_cb=handle_incoming)
session.send_chunk(msrp_transport.make_chunk(data='Hello world!'))
```

XCAP API

XCAP protocol allows a client to read, write, and modify application configuration data stored in XML format on a server. XCAP maps XML document sub-trees and element attributes to HTTP URIs, so that these components can be directly accessed by clients using HTTP protocol. An XCAP server is used by XCAP clients to store data like buddy lists and presence policy in combination with a SIP Presence server that supports PUBLISH, SUBSCRIBE and NOTIFY methods to provide a complete SIP SIMPLE solution.

XCAP client is implemented by python-xcaplib. The library provides xcaplib.client.XCAPClient class which is an HTTP client with an interface better suited for XCAP servers. The library also provides a version of XCAPClient (xcaplib.green.XCAPClient) built on top of eventlet, which may be used in twisted reactor.

Components

GET

```
get(self, application, node=None, etag=None, headers=None)
```

Make an HTTP GET request to the resource identified by application and node. Return a Resource instance on success. Raise HTTPError if the operation was unsuccessful.

PUT

```
put(self, application, resource, node=None, etag=None,
headers=None)
```

Make an HTTP PUT request to the resource identified by application and node. Use resource as a request body. Raise HTTPError is the operation was unsuccessful.

DELETE

```
delete(self, application, node=None, etag=None, headers=None)
```

Make an HTTP DELETE request to the resource identified by application and node. Raise HTTPError if the operation was unsuccessful.

Usage

```
client = XCAPClient(xcap_root, xcap_user_id, password=password)
document = file('examples/resource-lists.xml').read()

# put the document on the server
client.put('resource-lists', document)

# read the document from the server
got = client.get('resource-lists')

# get a specific element within a document
element = client.get('resource-lists', '/resource-
lists/list/entry/display-name')

# get an attribute:
res = client.get('resource-lists', '/resource-lists/list/entry/@uri')

# replace an element conditionally, based on the etag
client.put('resource-lists', '<entry
uri="sip:bob@example.com"><display-name>The Bob</display-
name></entry>',
           '/resource-lists/list/entry[@uri="sip:bob@example.com"]',
etag=stored_etag)

# delete an element
client.delete('resource-lists', node_selector, etag=res.etag)
```

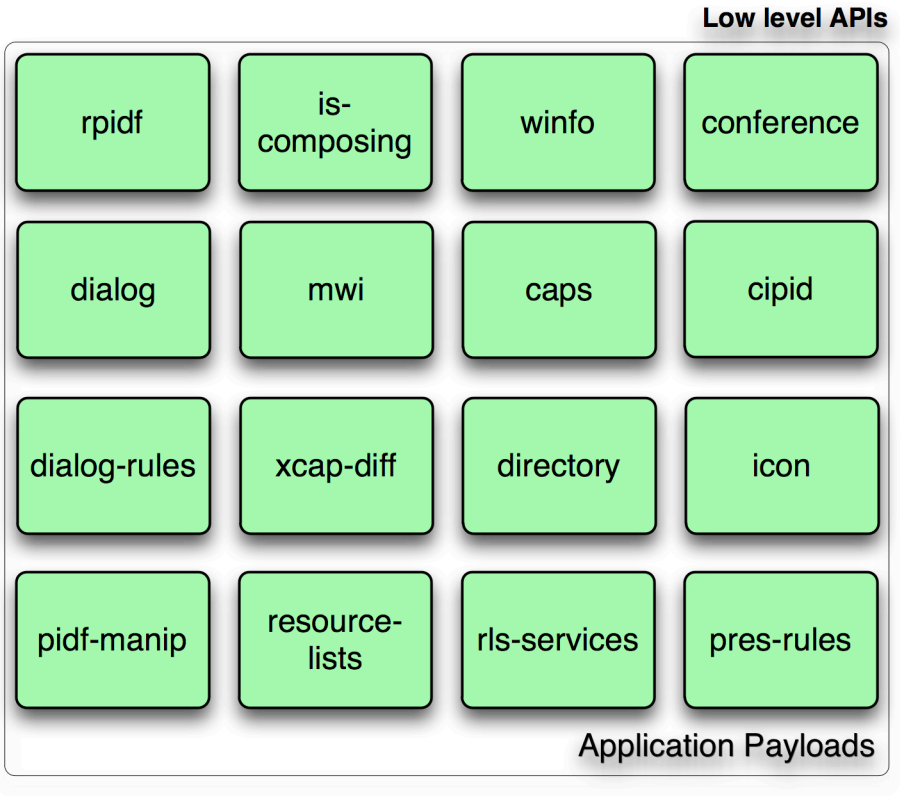
Payloads API

```
Implemented in sipsimple/payloads/*.py
```

The following modules are used for parsing and generating bodies carried using SIP PUBLISH/SUBSCRIBE/NOTIFY methods that have been designed for asynchronous event notifications to convey in real-time state and other information between end-points.

An example of state information is presence, which in its basic form provides user availability information based on end-user choice. In its advanced form, presence can provide rich state information including but not limited to user mood, geo-location, environment, noise level and type of communication desired. The information can be disseminated based on a granular policy which allows end-users to decide who has access to which part of the published information.

These applications are used by the SIP core Publication and Subscription classes.



Common Policy

```
Implemented in sipsimple/payloads/policy.py
```

Generic data types to be used in policy applications, according to RFC 4745.

Example

```
>>> alice = IdentityOne('sip:alice@example.com')
>>> carol = IdentityOne('tel:+1-212-555-1234')
>>> bob = IdentityOne('mailto:bob@example.net')
>>> print carol
tel:+1-212-555-1234
>>> id = Identity([alice, bob])
>>> print id
Identity([IdentityOne('sip:alice@example.com'),
IdentityOne('mailto:bob@example.net')])
>>> id[1:1] = [carol]
>>> print id
Identity([IdentityOne('sip:alice@example.com'), IdentityOne('tel:+1-
212-555-1234'), IdentityOne('mailto:bob@example.net')])
>>> conditions = Conditions([id])
>>> rule = Rule(id='f3g44r1', conditions=conditions, actions=Actions(),
transformations=Transformations())
>>> ruleset = RuleSet()
>>> ruleset.append(rule)
>>> print ruleset.toxml(pretty_print=True)
<?xml version='1.0' encoding='UTF-8'?>
<cp:ruleset xmlns:cp="urn:ietf:params:xml:ns:common-policy">
  <cp:rule id="f3g44r1">
    <cp:conditions>
      <cp:identity>
        <cp:one id="sip:alice@example.com"/>
        <cp:one id="mailto:bob@example.net"/>
        <cp:one id="tel:+1-212-555-1234"/>
      </cp:identity>
    </cp:conditions>
    <cp:actions/>
    <cp:transformations/>
  </cp:rule>
</cp:ruleset>
<BLANKLINE>
```

Pres-rules

```
Implemented in sipsimple/payloads/presrules.py
```

Parses and produces Presence Authorization Rules documents according to RFC 5025.

Authorization rules are stored on the XCAP server. The presence rules are generated either based on user initiative or as a response to a new subscription signaled by a change in the watcherinfo application.

Example

```
>>> conditions =
Conditions([Identity([IdentityOne('sip:user@example.com')])])
>>> actions = Actions([SubHandling('allow')])
>>> transformations = Transformations()
>>> psrv = ProvideServices(provides=[ServiceURIScheme('sip'),
ServiceURIScheme('mailto')])
>>> ppers = ProvidePersons(all=True)
>>> transformations[0:0] = [psrv, ppers]
>>> transformations.append(ProvideActivities('true'))
>>> transformations.append(ProvideUserInput('bare'))
>>> transformations.append(ProvideUnknownAttribute(ns='urn:vendor-
specific:foo-namespace', name='foo', value='true'))
>>> rule = Rule(id='a', conditions=conditions, actions=actions,
transformations=transformations)
>>> prules = PresRules([rule])
>>> print prules.toxml(pretty_print=True)
<?xml version='1.0' encoding='UTF-8'?>
<cp:ruleset xmlns:pr="urn:ietf:params:xml:ns:pres-rules"
xmlns:cp="urn:ietf:params:xml:ns:common-policy">
  <cp:rule id="a">
    <cp:conditions>
      <cp:identity>
        <cp:one id="sip:user@example.com"/>
      </cp:identity>
    </cp:conditions>
    <cp:actions>
      <pr:sub-handling>allow</pr:sub-handling>
    </cp:actions>
    <cp:transformations>
      <pr:provide-services>
        <pr:service-uri-scheme>sip</pr:service-uri-scheme>
        <pr:service-uri-scheme>mailto</pr:service-uri-scheme>
      </pr:provide-services>
      <pr:provide-persons>
        <pr:all-persons/>
      </pr:provide-persons>
      <pr:provide-activities>true</pr:provide-activities>
      <pr:provide-user-input>bare</pr:provide-user-input>
      <pr:provide-unknown-attribute ns="urn:vendor-specific:foo-
namespace" name="foo">true</pr:provide-unknown-attribute>
    </cp:transformations>
```

```
   </cp:rule>
</cp:ruleset>
<BLANKLINE>
```

Resource Lists

```
Implemented in sipsimple/payloads/resourcelists.py
```

This module provides convenient classes to parse and generate resource-lists documents as described in RFC 4826.

Used for server side storage of presence related buddy lists using XCAP protocol. The SIP clients maintain the resource-lists on the XCAP server which provides persisten storage and aggregation point for multiple devices.

Generation

```
>>> bill = Entry('sip:bill@example.com', display_name = 'Bill Doe')
>>> petri = EntryRef('some/ref')
>>> friends = List([bill, petri])
>>> rl = ResourceLists([friends])
>>> print rl.toxml(pretty_print=True)
<?xml version='1.0' encoding='UTF-8'?>
<rl:resource-lists xmlns:rl="urn:ietf:params:xml:ns:resource-lists">
  <rl:list>
    <rl:entry uri="sip:bill@example.com">
      <rl:display-name>Bill Doe</rl:display-name>
    </rl:entry>
    <rl:entry-ref ref="some/ref"/>
  </rl:list>
</rl:resource-lists>
<BLANKLINE>
```

toxml() wraps etree.tostring() and accepts all its arguments (like pretty_print).

Parsing

```
>>> r = ResourceLists.parse(example_from_section_3_3_rfc)
>>> len(r)
1

>>> friends = r[0]
>>> friends.name
'friends'

>>> bill = friends[0]
>>> bill.uri
'sip:bill@example.com'
>>> print bill.display_name
Bill Doe

>>> close_friends = friends[2]
>>> print close_friends[0]
"Joe Smith" <sip:joe@example.com>
>>> print close_friends[2].display_name
Marketing
```

RLS Services

```
Implemented in sipsimple/payloads/rlsservices.py
```

Parses and builds application/rls-services+xml documents according to RFC 4826.

Used for delegating presence related works to the server. The client build rls-services lists with buddies and instructs the server to subscribe to the sip uris indicated in the lists. This way the client can save bandwidth as the server performs the signaling for subscription and collection of notifications and provides consolidated answers to the SIP user agent.

Generation

```
>>> buddies = Service('sip:mybuddies@example.com',
'http://xcap.example.com/xxx', ['presence'])
>>> marketing = Service('sip:marketing@example.com')
>>> marketing.list = RLSList([Entry('sip:joe@example.com'),
Entry('sip:sudhir@example.com')])
>>> marketing.packages = ['presence']
>>> rls = RLSServices([buddies, marketing])
>>> print rls.toxml(pretty_print=True)
<?xml version='1.0' encoding='UTF-8'?>
<rls-services xmlns:rl="urn:ietf:params:xml:ns:resource-lists"
xmlns="urn:ietf:params:xml:ns:rls-services">
  <service uri="sip:mybuddies@example.com">
    <resource-list>http://xcap.example.com/xxx</resource-list>
    <packages>
      <package>presence</package>
    </packages>
  </service>
  <service uri="sip:marketing@example.com">
    <list>
      <rl:entry uri="sip:joe@example.com"/>
      <rl:entry uri="sip:sudhir@example.com"/>
    </list>
    <packages>
      <package>presence</package>
    </packages>
  </service>
</rls-services>
<BLANKLINE>
```

Parsing

```
>>> rls = RLSServices.parse(example_from_section_4_3_rfc)
>>> len(rls)
2

>>> rls[0].uri
'sip:mybuddies@example.com'

>>> print rls[0].list
```

```
http://xcap.example.com/xxx

>>> print rls[0].packages[0]
presence


>>> rls[1].uri
'sip:marketing@example.com'

>>> assert len(rls[1].packages) == 1 and rls[1].packages[0] ==
'presence'
```

Presence Data Model

```
Implemented in sipsimple/payloads/presdm.py
```

PIDF handling according to RFC 3863 and RFC 3379. This module provides classes to parse and generate PIDF documents.

Used to parse NOTIFY body for presence event and generate state information for use with PUBLISH method and to parse the state of buddy lists entries we have subscribed to. A SIP client typically instantiates a new PIDF object for itself and for each buddy it SUBSCRIBEs to and updates each object when a NOTIFY is received. The list of buddies is maintained using the resource-lists XCAP application.

Example

```
>>> from datetime import datetime
>>> pidf = PIDF('pres:someone@example.com')
>>> status = Status(basic=Basic('open'))
>>> contact = Contact('im:someone@mobilecarrier.net')
>>> contact.priority = "0.8"
>>> tuple1 = Service('bs35r9', notes=[ServiceNote("Don't Disturb
Please!"), ServiceNote("Ne derangez pas, s'il vous plait", lang="fr")],
status=status)
>>> tuple1.contact = contact
>>> tuple1.timestamp = Timestamp(datetime(2008, 9, 11, 20, 42, 03))
>>> tuple2 = Service('eg92n8', status=Status(basic=Basic('open')),
contact=Contact('mailto:someone@example.com'))
>>> tuple2.contact.priority = "1.0"
>>> pidf.notes.add(Note("I'll be in Tokyo next week"))
>>> pidf.append(tuple1)
>>> pidf.append(tuple2)
>>> print pidf.toxml(pretty_print=True)
<?xml version='1.0' encoding='UTF-8'?>
<presence xmlns:rpid="urn:ietf:params:xml:ns:pidf:rpid"
xmlns:dm="urn:ietf:params:xml:ns:pidf:data-model"
xmlns="urn:ietf:params:xml:ns:pidf" entity="pres:someone@example.com"
  <tuple id="bs35r9">
    <status>
      <basic>open</basic>
    </status>
    <contact priority="0.8">im:someone@mobilecarrier.net</contact>
    <note>Don't Disturb Please!</note>
    <note xml:lang="fr">Ne derangez pas, s'il vous plait</note>
    <timestamp>2008-09-11T20:42:03Z</timestamp>
  </tuple>
  <tuple id="eg92n8">
    <status>
      <basic>open</basic>
    </status>
    <contact priority="1.0">mailto:someone@example.com</contact>
  </tuple>
  <note>I'll be in Tokyo next week</note>
</presence>
<BLANKLINE>
```

Rich Presence Extension

```
Implemented in sipsimple/payloads/rpid.py
```

RPID handling according to RFC 4480. This module provides an extension to PIDF (module sipsimple.applications.presdm) to support rich presence.

```
__all__ = ['_rpid_namespace_',
           'ActivityElement',
           'MoodElement',
           'PlaceTypeElement',
           'PrivacyElement',
           'SphereElement',
           'RPIDNote',
           'Activities',
           'Mood',
           'PlaceIs',
           'AudioPlaceInformation',
           'VideoPlaceInformation',
           'TextPlaceInformation',
           'PlaceType',
           'AudioPrivacy',
           'TextPrivacy',
           'VideoPrivacy',
           'Privacy',
           'Relationship',
           'ServiceClass',
           'Sphere',
           'StatusIcon',
           'TimeOffset',
           'UserInput',
           'Class',
           'Other']
```

Watcher-info

```
Implemented in sipsimple/payloads/watcherinfo.py
```

Parses application/watcherinfo+xml documents according to RFC 3857 and RFC3858.

Used for parsing of NOTIFY body for presence.winfo event. Used for keeping track of watchers that subscribed to our presentity. Based on this information the authorization rules can be managed using presrules.py. To retrieve this information the SIP client must subscribe to its own address for event presence.winfo.

Example

```
>>> winfo_doc='''<?xml version="1.0"?>
... <watcherinfo xmlns="urn:ietf:params:xml:ns:watcherinfo"
...              version="0" state="full">
...   <watcher-list resource="sip:professor@example.net"
package="presence">
...     <watcher status="active"
...              id="8ajksjda7s"
...              duration-subscribed="509"
...              event="approved" >sip:userA@example.net</watcher>
...     <watcher status="pending"
...              id="hh8juja87s997-ass7"
...              display-name="Mr. Subscriber"
...              event="subscribe">sip:userB@example.org</watcher>
...   </watcher-list>
... </watcherinfo>'''
>>> winfo = WatcherInfo()
```

The return value of winfo.update() is a dictionary containing WatcherList objects as keys and lists of the updated watchers as values.

```
>>> updated = winfo.update(winfo_doc)
>>> len(updated['sip:professor@example.net'])
2
```

winfo.pending, winfo.terminated and winfo.active are dictionaries indexed by WatcherList objects as keys and lists of Wacher objects as values.

```
>>> print winfo.pending['sip:professor@example.net'][0]
"Mr. Subscriber" <sip:userB@example.org>
>>> print winfo.pending['sip:professor@example.net'][1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> print winfo.active['sip:professor@example.net'][0]
sip:userA@example.net
>>> len(winfo.terminated['sip:professor@example.net'])
```

```
0
```

winfo.wlists is the list of WatcherList objects

```
>>> list(winfo.wlists[0].active) ==
list(winfo.active['sip:professor@example.net'])
True
```

See the classes for more information.

XCAP-diff

```
Implemented in sipsimple/payloads/xcapdiff.py
```

This module allows parsing and building xcap-diff documents according to draft-ietf-simple-xcap-diff.

Used to parse NOTIFY body for xcap-diff event. Used to detect changes in XCAP documents changed by other device configured for the same presentity.

Is-composing

```
Implemented in sipsimple/payloads/iscomposing.py
```

This module parses and produces isComposing messages according to RFC3994.

Message Summary

```
Implemented in sipsimple/payloads/messagesummary.py
```

This module parses and produces message-summary messages according to RF3842.

User Agent Capability

```
Implemented in python-sipsimple/payloads/caps.py
```

User Agent Capability Extension handling according to RFC5196

This module provides an extension to PIDF to describe a user-agent capabilities in the PIDF documents.

```
__all__ = ['caps_namespace',
           'Audio',
           'Application',
           'Data',
           'Control',
           'Video',
           'Video',
           'Text',
           'Message',
           'Type',
           'Automata',
           'Class',
           'ClassPersonal',
           'ClassBusiness',
           'Duplex',
           'DuplexFull',
           'DuplexHalf',
           'DuplexReceiveOnly',
           'DuplexSendOnly',
           'Description',
           'EventPackages',
           'EventConference',
           'EventDialog',
           'EventKpml',
           'EventMessageSummary',
           'EventPocSettings',
           'EventPresence',
           'EventReg',
           'EventRefer',
           'EventSiemensRtpStats',
           'EventSpiritsIndps',
           'EventSpiritsUserProf',
           'EventWinfo',
           'Priority',
           'PriorityLowerthan',
           'PriorityHigherthan',
           'PriorityEquals',
           'PriorityRange',
           'Methods',
           'MethodAck',
           'MethodBye',
           'MethodCancel',
           'MethodInfo',
           'MethodInvite',
           'MethodMessage',
           'MethodNotify',
```

```
            'MethodOptions',
            'MethodPrack',
            'MethodPublish',
            'MethodRefer',
            'MethodRegister',
            'MethodSubscribe',
            'MethodUpdate',
            'Extensions',
            'ExtensionRel100',
            'ExtensionEarlySession',
            'ExtensionEventList',
            'ExtensionFromChange',
            'ExtensionGruu',
            'ExtensionHistinfo',
            'ExtensionJoin',
            'ExtensionNoRefSub',
            'ExtensionPath',
            'ExtensionPrecondition',
            'ExtensionPref',
            'ExtensionPrivacy',
            'ExtensionRecipientListInvite',
            'ExtensionRecipientListSubscribe',
            'ExtensionReplaces',
            'ExtensionResourcePriority',
            'ExtensionSdpAnat',
            'ExtensionSecAgree',
            'ExtensionTdialog',
            'ExtensionTimer',
            'Schemes',
            'Scheme',
            'Actor',
            'ActorPrincipal',
            'ActorAttendant',
            'ActorMsgTaker',
            'ActorInformation',
            'IsFocus',
            'Languages',
            'Language',
            'Servcaps',
            'Mobility',
            'MobilityFixed',
            'MobilityMobile',
            'Devcaps',
            'ServcapsExtension',
            'EventPackagesExtension',
            'PriorityExtension',
            'MethodsExtension',
            'ExtensionsExtension',
            'DevcapsExtension',
            'MobilityExtension']
```

CIPID

```
Implemented in python-sipsimple/payloads/cipid.py
```

CIPID handling according to RFC4482. This module provides an extension to PIDF to provide additional contact information about a presentity.

```
__all__ = ['cipid_namespace', 'Card', 'DisplayName', 'Homepage',
'Icon', 'Map', 'Sound']
```

Conference

```
Implemented in python-sipsimple/payloads/conference.py
```

Parses and produces conference-info messages according to RFC4575.

```
__all__ = ['namespace',
           'ConferenceApplication',
           'ConferenceDescription',
           'ConfUris',
           'ConfUrisEntry',
           'ServiceUris',
           'ServiceUrisEntry',
           'UrisTypeModified',
           'UrisTypeEntry',
           'AvailableMedia',
           'AvailableMediaEntry',
           'Users',
           'User',
           'AssociatedAors',
           'Roles',
           'Role',
           'Endpoint',
           'CallInfo',
           'Sip',
           'Referred',
           'JoiningInfo',
           'DisconnectionInfo',
           'HostInfo',
           'HostInfoUris',
           'ConferenceState',
           'SidebarsByRef',
           'SidebarsByVal',
           'Conference',
           'ConferenceDescriptionExtension']
```

Dialog Info

```
Implemented in python-sipsimple/payloads/dialoginfo.py
```

Parses and produces conference-info messages according to RFC4575.

```
__all__ = ['namespace',
           'DialogInfoApplication',
           'DialogState',
           'Replaces',
           'ReferredBy',
           'Identity',
           'Param',
           'Target',
           'Local',
           'Remote',
           'Dialog',
           'DialogInfo']
```

Sample Code

```
Implemented in sipclients/sip-session.py
```

sip-session command line script is a show-case for the powerful features of SIP SIMPLE development kit related to establishing, modifying and terminating SIP sessions with multiple media types like VoIP, Instant Messaging and File Transfer.

```
#!/usr/bin/env python
# Copyright (C) 2008-2010 AG Projects. See LICENSE for details.
#

import hashlib
import os
import re
import signal

from datetime import datetime
from itertools import chain
from operator import attrgetter
from optparse import OptionParser
from threading import Event, Thread
from time import sleep

from application import log
from application.notification import IObserver, NotificationCenter
from application.python.queue import EventQueue
from application.python.util import Null
from eventlet import api, proc
from twisted.internet import reactor
from zope.interface import implements

from sipsimple.core import Engine, SIPCoreError, SIPURI, ToHeader

from sipsimple.account import Account, AccountManager, BonjourAccount
from sipsimple.application import SIPApplication
from sipsimple.audio import WavePlayer
from sipsimple.configuration import ConfigurationError
from sipsimple.configuration.backend.file import FileBackend
from sipsimple.configuration.settings import SIPSimpleSettings
from sipsimple.lookup import DNSLookup
from sipsimple.session import IllegalStateError, Session
from sipsimple.streams import AudioStream, ChatStream, FileSelector, FileTransferStream
from sipsimple.util import run_in_green_thread

from sipclient.configuration import config_filename
from sipclient.configuration.account import AccountExtension
from sipclient.configuration.datatypes import ResourcePath
from sipclient.configuration.settings import SIPSimpleSettingsExtension
from sipclient.log import Logger
from sipclient.system import IPAddressMonitor
from sipclient.ui import Prompt, Question, RichText, UI


# This is a helper function for sending formatted notice messages
def send_notice(text, bold=True):
    ui = UI()
    if isinstance(text, list):
        ui.writelines([RichText(line, bold=bold) if not isinstance(line, RichText) else line for line
in text])
    elif isinstance(text, RichText):
        ui.write(text)
    else:
        ui.write(RichText(text, bold=bold))


# Utility classes
#

class BonjourNeighbour(object):
    def __init__(self, uri, display_name=None):
        self.uri = uri
        self.display_name = display_name

    def __hash__(self):
        return hash(self.uri)
```

```
    def __eq__(self, other):
        if isinstance(other, BonjourNeighbour):
            return self.uri == other.uri and self.display_name == other.display_name
        else:
            return self.uri == other

    def __ne__(self, other):
        return not self.__eq__(other)


class RTPStatisticsThread(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.setDaemon(True)
        self.stopped = False

    def run(self):
        application = SIPSessionApplication()
        while not self.stopped:
            if application.active_session is not None and application.active_session.streams:
                try:
                    audio_stream = [stream for stream in application.active_session.streams if
isinstance(stream, AudioStream)][0]
                except IndexError:
                    pass
                else:
                    stats = audio_stream.statistics
                    if stats is not None:
                        reactor.callFromThread(send_notice, '%s RTP statistics: RTT=%d ms, packet
loss=%.1f%%, jitter RX/TX=%d/%d ms' %

(datetime.now().replace(microsecond=0),
                                                                    stats['rtt']['avg'] / 1000,
                                                                    100.0 * stats['rx']['packets_lost'] /
stats['rx']['packets'] if stats['rx']['packets'] else 0,
                                                                    stats['rx']['jitter']['avg'] / 1000,
                                                                    stats['tx']['jitter']['avg'] / 1000))
            sleep(10)

    def stop(self):
        self.stopped = True


class NATDetector(object):
    implements(IObserver)

    def __init__(self):
        notification_center = NotificationCenter()
        notification_center.add_observer(self, name='SIPApplicationDidStart')

    def handle_notification(self, notification):
        handler = getattr(self, '_NH_%s' % notification.name, Null())
        handler(notification)

    def _NH_SIPApplicationDidStart(self, notification):
        notification_center = NotificationCenter()
        lookup = DNSLookup()
        notification_center.add_observer(self, name='SIPEngineDetectedNATType')
        notification_center.add_observer(self, sender=lookup)
        lookup.lookup_service(SIPURI(host=notification.sender.account.id.domain), 'stun')

    def _NH_SIPEngineDetectedNATType(self, notification):
        if notification.data.succeeded:
            send_notice('Detected NAT type: %s' % notification.data.nat_type)

    def _NH_DNSLookupDidSucceed(self, notification):
        engine = Engine()
        stun_server, stun_port = notification.data.result[0]
        engine.detect_nat_type(stun_server, stun_port)


class OutgoingCallInitializer(object):
    implements(IObserver)

    def __init__(self, account, target, audio=False, chat=False):
        self.account = account
        self.target = target
        self.streams = []
        if audio:
            self.streams.append(AudioStream(account))
        if chat:
```

```
            self.streams.append(ChatStream(account))
        self.wave_ringtone = None

    def start(self):
        if isinstance(self.account, BonjourAccount) and '@' not in self.target:
            send_notice('Bonjour mode requires a host in the destination address')
            return
        if '@' not in self.target:
            self.target = '%s@%s' % (self.target, self.account.id.domain)
        if not self.target.startswith('sip:') and not self.target.startswith('sips:'):
            self.target = 'sip:' + self.target
        try:
            self.target = SIPURI.parse(self.target)
        except SIPCoreError:
            send_notice('Illegal SIP URI: %s' % self.target)
        else:
            if '.' not in self.target.host and not isinstance(self.account, BonjourAccount):
                self.target.host = '%s.%s' % (self.target.host, self.account.id.domain)
            lookup = DNSLookup()
            notification_center = NotificationCenter()
            notification_center.add_observer(self, sender=lookup)
            settings = SIPSimpleSettings()
            if isinstance(self.account, Account) and self.account.sip.outbound_proxy is not None:
                uri = SIPURI(host=self.account.sip.outbound_proxy.host,
port=self.account.sip.outbound_proxy.port, parameters={'transport':
self.account.sip.outbound_proxy.transport})
            else:
                uri = self.target
            lookup.lookup_sip_proxy(uri, settings.sip.transport_list)

    def handle_notification(self, notification):
        handler = getattr(self, '_NH_%s' % notification.name, Null())
        handler(notification)

    def _NH_DNSLookupDidSucceed(self, notification):
        notification_center = NotificationCenter()
        notification_center.remove_observer(self, sender=notification.sender)
        session = Session(self.account)
        notification_center.add_observer(self, sender=session)
        session.connect(ToHeader(self.target), routes=notification.data.result, streams=self.streams)
        application = SIPSessionApplication()
        application.outgoing_session = session

    def _NH_DNSLookupDidFail(self, notification):
        send_notice('Call to %s failed: DNS lookup error: %s' % (self.target,
notification.data.error))
        notification_center = NotificationCenter()
        notification_center.remove_observer(self, sender=notification.sender)

    def _NH_SIPSessionNewOutgoing(self, notification):
        session = notification.sender
        local_identity = str(session.local_identity.uri)
        if session.local_identity.display_name:
            local_identity = '"%s" <%s>' % (session.local_identity.display_name, local_identity)
        remote_identity = str(session.remote_identity.uri)
        if session.remote_identity.display_name:
            remote_identity = '"%s" <%s>' % (session.remote_identity.display_name, remote_identity)
        send_notice("Initiating SIP session from '%s' to '%s' via %s..." % (local_identity,
remote_identity, session.route))

    def _NH_SIPSessionGotRingIndication(self, notification):
        settings = SIPSimpleSettings()
        ui = UI()
        ringtone = settings.sounds.audio_outbound
        if ringtone:
            self.wave_ringtone = WavePlayer(SIPApplication.voice_audio_mixer,
ringtone.path.normalized, volume=ringtone.volume, loop_count=0, pause_time=2)
            SIPApplication.voice_audio_bridge.add(self.wave_ringtone)
            self.wave_ringtone.start()
        ui.status = 'Ringing...'

    def _NH_SIPSessionWillStart(self, notification):
        ui = UI()
        if self.wave_ringtone:
            self.wave_ringtone.stop()
            SIPApplication.voice_audio_bridge.remove(self.wave_ringtone)
            self.wave_ringtone = None
        ui.status = 'Connecting...'

    def _NH_SIPSessionDidStart(self, notification):
        notification_center = NotificationCenter()
        ui = UI()
```

```
        session = notification.sender
        notification_center.remove_observer(self, sender=session)
        ui.status = 'Connected'
        reactor.callLater(2, setattr, ui, 'status', None)

        application = SIPSessionApplication()
        application.outgoing_session = None

        for stream in notification.data.streams:
            if isinstance(stream, AudioStream):
                send_notice('Audio session established using "%s" codec at %sHz' % (stream.codec,
stream.sample_rate))
                if stream.ice_active:
                    send_notice('Audio RTP endpoints %s:%d (ICE type %s) <-> %s:%d (ICE type %s)' %
(stream.local_rtp_address, stream.local_rtp_port, stream.local_rtp_candidate_type,
stream.remote_rtp_address, stream.remote_rtp_port, stream.remote_rtp_candidate_type))
                else:
                    send_notice('Audio RTP endpoints %s:%d <-> %s:%d' % (stream.local_rtp_address,
stream.local_rtp_port, stream.remote_rtp_address, stream.remote_rtp_port))
                if stream.srtp_active:
                    send_notice('RTP audio stream is encrypted')
        if session.remote_user_agent is not None:
            send_notice('Remote SIP User Agent is "%s"' % session.remote_user_agent)

    def _NH_SIPSessionDidFail(self, notification):
        notification_center = NotificationCenter()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)

        ui = UI()
        ui.status = None

        application = SIPSessionApplication()
        application.outgoing_session = None

        if self.wave_ringtone:
            self.wave_ringtone.stop()
            SIPApplication.voice_audio_bridge.remove(self.wave_ringtone)
            self.wave_ringtone = None
        if notification.data.failure_reason == 'user request' and notification.data.code == 487:
            send_notice('SIP session cancelled')
        elif notification.data.failure_reason == 'user request':
            send_notice('SIP session rejected by user (%d %s)' % (notification.data.code,
notification.data.reason))
        else:
            send_notice('SIP session failed: %s' % notification.data.failure_reason)


class IncomingCallInitializer(object):
    implements(IObserver)

    sessions = 0
    tone_ringtone = None

    def __init__(self, session, auto_answer_interval=None):
        self.session = session
        self.auto_answer_interval = auto_answer_interval
        self.question = None

    def start(self):
        IncomingCallInitializer.sessions += 1
        notification_center = NotificationCenter()
        notification_center.add_observer(self, sender=self.session)

        # start auto-answer
        self.answer_timer = None
        if self.auto_answer_interval == 0:
            self.session.accept(self.session.proposed_streams)
            return
        elif self.auto_answer_interval > 0:
            self.answer_timer = reactor.callFromThread(reactor.callLater, self.auto_answer_interval,
self.session.accept, self.session.proposed_streams)

        # start ringing
        application = SIPSessionApplication()
        self.wave_ringtone = None
        if application.active_session is None:
            if IncomingCallInitializer.sessions == 1:
                ringtone = self.session.account.sounds.audio_inbound.sound_file if
self.session.account.sounds.audio_inbound is not None else None
                if ringtone:
```

```
                    self.wave_ringtone = WavePlayer(SIPApplication.alert_audio_mixer,
ringtone.path.normalized, volume=ringtone.volume, loop_count=0, pause_time=2)
                    SIPApplication.alert_audio_bridge.add(self.wave_ringtone)
                    self.wave_ringtone.start()
        elif IncomingCallInitializer.tone_ringtone is None:
            IncomingCallInitializer.tone_ringtone = WavePlayer(SIPApplication.voice_audio_mixer,
ResourcePath('sounds/ring_tone.wav').normalized, loop_count=0, pause_time=6)
            SIPApplication.voice_audio_bridge.add(IncomingCallInitializer.tone_ringtone)
            IncomingCallInitializer.tone_ringtone.start()
        self.session.send_ring_indication()

        # ask question
        identity = str(self.session.remote_identity.uri)
        if self.session.remote_identity.display_name:
            identity = '"%s" <%s>' % (self.session.remote_identity.display_name, identity)
        streams = '/'.join(stream.type for stream in self.session.proposed_streams)
        self.question = Question("Incoming %s from '%s', do you want to accept?
(a)ccept/(r)eject/(b)usy" % (streams, identity), 'arbi', bold=True)
        notification_center.add_observer(self, sender=self.question)
        ui = UI()
        ui.add_question(self.question)

    def handle_notification(self, notification):
        handler = getattr(self, '_NH_%s' % notification.name, Null())
        handler(notification)

    def _NH_UIQuestionGotAnswer(self, notification):
        notification_center = NotificationCenter()
        ui = UI()
        notification_center.remove_observer(self, sender=notification.sender)
        answer = notification.data.answer
        self.question = None
        if answer == 'a':
            self.session.accept(self.session.proposed_streams)
            ui.status = 'Accepting...'
        elif answer == 'r':
            self.session.reject()
            ui.status = 'Rejecting...'
        elif answer == 'b':
            self.session.reject(486)
            ui.status = 'Sending Busy Here...'

        if self.wave_ringtone:
            self.wave_ringtone.stop()
            self.wave_ringtone = None
        if IncomingCallInitializer.sessions > 1:
            if IncomingCallInitializer.tone_ringtone is None:
                IncomingCallInitializer.tone_ringtone = WavePlayer(SIPApplication.voice_audio_mixer,
ResourcePath('sounds/ring_tone.wav').normalized, loop_count=0, pause_time=6)
                SIPApplication.voice_audio_bridge.add(IncomingCallInitializer.tone_ringtone)
                IncomingCallInitializer.tone_ringtone.start()
        elif IncomingCallInitializer.tone_ringtone:
            IncomingCallInitializer.tone_ringtone.stop()
            IncomingCallInitializer.tone_ringtone = None
        if self.answer_timer is not None and self.answer_timer.active():
            self.answer_timer.cancel()

    def _NH_SIPSessionWillStart(self, notification):
        ui = UI()
        if self.question is not None:
            notification_center = NotificationCenter()
            notification_center.remove_observer(self, sender=self.question)
            ui.remove_question(self.question)
            self.question = None
        ui.status = 'Connecting...'

    def _NH_SIPSessionDidStart(self, notification):
        notification_center = NotificationCenter()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)
        IncomingCallInitializer.sessions -= 1
        application = SIPSessionApplication()

        ui = UI()
        ui.status = 'Connected'
        reactor.callLater(2, setattr, ui, 'status', None)

        identity = str(session.remote_identity.uri)
        if session.remote_identity.display_name:
            identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
        send_notice("SIP session with '%s' established" % identity)
        for stream in notification.data.streams:
```

```
            if isinstance(stream, AudioStream):
                send_notice('Audio stream using "%s" codec at %sHz' % (stream.codec,
stream.sample_rate))
                if stream.ice_active:
                    send_notice('Audio RTP endpoints %s:%d (ICE type %s) <-> %s:%d (ICE type %s)' %
(stream.local_rtp_address, stream.local_rtp_port, stream.local_rtp_candidate_type,
stream.remote_rtp_address, stream.remote_rtp_port, stream.remote_rtp_candidate_type))
                else:
                    send_notice('Audio RTP endpoints %s:%d <-> %s:%d' % (stream.local_rtp_address,
stream.local_rtp_port, stream.remote_rtp_address, stream.remote_rtp_port))
                if stream.srtp_active:
                    send_notice('RTP audio stream is encrypted')
        if session.remote_user_agent is not None:
            send_notice('Remote SIP User Agent is "%s"' % session.remote_user_agent)

    def _NH_SIPSessionDidFail(self, notification):
        notification_center = NotificationCenter()
        ui = UI()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)

        ui.status = None

        if self.question is not None:
            notification_center.remove_observer(self, sender=self.question)
            ui.remove_question(self.question)
            self.question = None

        IncomingCallInitializer.sessions -= 1
        if self.wave_ringtone:
            self.wave_ringtone.stop()
            self.wave_ringtone = None
        if IncomingCallInitializer.sessions == 0 and IncomingCallInitializer.tone_ringtone is not
None:
            IncomingCallInitializer.tone_ringtone.stop()
            IncomingCallInitializer.tone_ringtone = None
        if notification.data.failure_reason == 'user request' and notification.data.code == 487:
            send_notice('SIP session cancelled by user')
        if notification.data.failure_reason == 'Call completed elsewhere' and notification.data.code
== 487:
            send_notice('SIP session cancelled, call was answered elsewhere')
        elif notification.data.failure_reason == 'user request':
            send_notice('SIP session rejected (%d %s)' % (notification.data.code,
notification.data.reason))
        else:
            send_notice('SIP session failed: %s' % notification.data.failure_reason)


class OutgoingProposalHandler(object):
    implements(IObserver)

    def __init__(self, session, audio=False, chat=False):
        self.session = session
        self.stream = None
        if audio:
            self.stream = AudioStream(session.account)
        if chat:
            self.stream = ChatStream(session.account)
        if not self.stream:
            raise ValueError("Need to specify exactly one stream")

    def start(self):
        notification_center = NotificationCenter()
        notification_center.add_observer(self, sender=self.session)
        try:
            self.session.add_stream(self.stream)
        except IllegalStateError:
            notification_center.remove_observer(self, sender=self.session)
            raise

        remote_identity = str(self.session.remote_identity.uri)
        if self.session.remote_identity.display_name:
            remote_identity = '"%s" <%s>' % (self.session.remote_identity.display_name,
remote_identity)
        send_notice("Proposing %s to '%s'..." % (self.stream.type, remote_identity))

    def handle_notification(self, notification):
        handler = getattr(self, '_NH_%s' % notification.name, Null())
        handler(notification)

    def _NH_SIPSessionGotAcceptProposal(self, notification):
        notification_center = NotificationCenter()
```

```
        notification_center.remove_observer(self, sender=self.session)
        application = SIPSessionApplication()
        application.sessions_with_proposals.remove(notification.sender)
        send_notice('Proposal accepted')

    def _NH_SIPSessionGotRejectProposal(self, notification):
        notification_center = NotificationCenter()
        notification_center.remove_observer(self, sender=self.session)
        application = SIPSessionApplication()
        application.sessions_with_proposals.remove(notification.sender)

        ui = UI()
        ui.status = None
        if notification.data.code == 487:
            send_notice('Proposal cancelled (%d %s)' % (notification.data.code,
notification.data.reason))
        else:
            send_notice('Proposal rejected (%d %s)' % (notification.data.code,
notification.data.reason))

    def _NH_SIPSessionDidEnd(self, notification):
        notification_center = NotificationCenter()
        notification_center.remove_observer(self, sender=self.session)


class IncomingProposalHandler(object):
    implements(IObserver)

    sessions = 0
    tone_ringtone = None

    def __init__(self, session):
        self.session = session
        self.question = None

    def start(self):
        IncomingProposalHandler.sessions += 1
        notification_center = NotificationCenter()
        notification_center.add_observer(self, sender=self.session)

        # start ringing
        if IncomingProposalHandler.tone_ringtone is None:
            IncomingProposalHandler.tone_ringtone = WavePlayer(SIPApplication.voice_audio_mixer,
ResourcePath('sounds/ring_tone.wav').normalized, loop_count=0, pause_time=6)
            SIPApplication.voice_audio_bridge.add(IncomingProposalHandler.tone_ringtone)
            IncomingProposalHandler.tone_ringtone.start()
        self.session.send_ring_indication()

        # ask question
        identity = str(self.session.remote_identity.uri)
        if self.session.remote_identity.display_name:
            identity = '"%s" <%s>' % (self.session.remote_identity.display_name, identity)
        streams = ', '.join(stream.type for stream in self.session.proposed_streams)
        self.question = Question("'%s' wants to add %s, do you want to accept? (a)ccept/(r)eject" %
(identity, streams), 'ar', bold=True)
        notification_center.add_observer(self, sender=self.question)
        ui = UI()
        ui.add_question(self.question)

    def handle_notification(self, notification):
        handler = getattr(self, '_NH_%s' % notification.name, Null())
        handler(notification)

    def _NH_UIQuestionGotAnswer(self, notification):
        notification_center = NotificationCenter()
        ui = UI()
        notification_center.remove_observer(self, sender=notification.sender)
        answer = notification.data.answer
        self.question = None
        if answer == 'a':
            self.session.accept_proposal(self.session.proposed_streams)
            ui.status = 'Accepting proposal...'
        elif answer == 'r':
            self.session.reject_proposal()
            ui.status = 'Rejecting proposal...'

        if IncomingProposalHandler.sessions == 1 and IncomingProposalHandler.tone_ringtone:
            IncomingProposalHandler.tone_ringtone.stop()
            IncomingProposalHandler.tone_ringtone = None

    def _NH_SIPSessionGotAcceptProposal(self, notification):
        notification_center = NotificationCenter()
```

```
        session = notification.sender
        notification_center.remove_observer(self, sender=session)
        application = SIPSessionApplication()
        application.sessions_with_proposals.remove(notification.sender)
        IncomingProposalHandler.sessions -= 1

        ui = UI()
        ui.status = None
        send_notice('Proposal accepted')

    def _NH_SIPSessionGotRejectProposal(self, notification):
        notification_center = NotificationCenter()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)
        application = SIPSessionApplication()
        application.sessions_with_proposals.remove(notification.sender)
        IncomingProposalHandler.sessions -= 1

        ui = UI()
        ui.status = None
        if notification.data.code == 487:
            send_notice('Proposal cancelled (%d %s)' % (notification.data.code,
notification.data.reason))
        else:
            send_notice('Proposal rejected (%d %s)' % (notification.data.code,
notification.data.reason))

        if IncomingProposalHandler.tone_ringtone:
            IncomingProposalHandler.tone_ringtone.stop()
            IncomingProposalHandler.tone_ringtone = None

        if self.question is not None:
            notification_center.remove_observer(self, sender=self.question)
            ui.remove_question(self.question)
            self.question = None

    def _NH_SIPSessionHadProposalFailure(self, notification):
        notification_center = NotificationCenter()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)
        IncomingProposalHandler.sessions -= 1

        ui = UI()
        ui.status = None
        send_notice('Proposal failed (%s)' % notification.data.failure_reason)

    def _NH_SIPSessionDidEnd(self, notification):
        notification_center = NotificationCenter()
        ui = UI()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)

        ui.status = None

        if self.question is not None:
            notification_center.remove_observer(self, sender=self.question)
            ui.remove_question(self.question)
            self.question = None

        IncomingProposalHandler.sessions -= 1
        if IncomingProposalHandler.sessions == 0 and IncomingProposalHandler.tone_ringtone is not
None:
            IncomingProposalHandler.tone_ringtone.stop()
            IncomingProposalHandler.tone_ringtone = None


class OutgoingTransferHandler(object):
    implements(IObserver)

    def __init__(self, account, target, filepath):
        self.account = account
        self.target = target
        self.filepath = filepath
        self.file_selector = None
        self.finished = False
        self.hash_compute_proc = None
        self.session = None
        self.wave_ringtone = None

    @run_in_green_thread
    def start(self):
        if isinstance(self.account, BonjourAccount) and '@' not in self.target:
```

```
            send_notice('Bonjour mode requires a host in the destination address')
            return
        if '@' not in self.target:
            self.target = '%s@%s' % (self.target, self.account.id.domain)
        if not self.target.startswith('sip:') and not self.target.startswith('sips:'):
            self.target = 'sip:' + self.target
        try:
            self.target = SIPURI.parse(self.target)
        except SIPCoreError:
            send_notice('Illegal SIP URI: %s' % self.target)
        else:
            send_notice('Computing hash...')
            def compute_hash():
                try:
                    self.file_selector = FileSelector.for_file(self.filepath)
                except Exception, e:
                    send_notice('Failed to read file "%s": %s' % (self.filepath, e))
            self.hash_compute_proc = proc.spawn(compute_hash)

            if '.' not in self.target.host and not isinstance(self.account, BonjourAccount):
                self.target.host = '%s.%s' % (self.target.host, self.account.id.domain)
            lookup = DNSLookup()
            notification_center = NotificationCenter()
            notification_center.add_observer(self, sender=lookup)
            settings = SIPSimpleSettings()
            if isinstance(self.account, Account) and self.account.sip.outbound_proxy is not None:
                uri = SIPURI(host=self.account.sip.outbound_proxy.host,
port=self.account.sip.outbound_proxy.port, parameters={'transport':
self.account.sip.outbound_proxy.transport})
            else:
                uri = self.target
            lookup.lookup_sip_proxy(uri, settings.sip.transport_list)

    def handle_notification(self, notification):
        handler = getattr(self, '_NH_%s' % notification.name, Null())
        handler(notification)

    def _NH_DNSLookupDidSucceed(self, notification):
        notification_center = NotificationCenter()
        notification_center.remove_observer(self, sender=notification.sender)

        self.hash_compute_proc.wait()
        if self.file_selector is None:
            return

        self.session = Session(self.account)
        notification_center.add_observer(self, sender=self.session)
        self.session.connect(ToHeader(self.target), routes=notification.data.result,
streams=[FileTransferStream(self.account, self.file_selector)])

    def _NH_DNSLookupDidFail(self, notification):
        send_notice('File transfer to %s failed: DNS lookup error: %s' % (self.target,
notification.data.error))
        notification_center = NotificationCenter()
        notification_center.remove_observer(self, sender=notification.sender)

    def _NH_SIPSessionNewOutgoing(self, notification):
        session = notification.sender
        local_identity = str(session.local_identity.uri)
        if session.local_identity.display_name:
            local_identity = '"%s" <%s>' % (session.local_identity.display_name, local_identity)
        remote_identity = str(session.remote_identity.uri)
        if session.remote_identity.display_name:
            remote_identity = '"%s" <%s>' % (session.remote_identity.display_name, remote_identity)
        send_notice("Initiating file transfer from '%s' to '%s' via %s..." % (local_identity,
remote_identity, session.route))

    def _NH_SIPSessionGotRingIndication(self, notification):
        settings = SIPSimpleSettings()
        ui = UI()
        ringtone = settings.sounds.audio_outbound
        if ringtone:
            self.wave_ringtone = WavePlayer(SIPApplication.voice_audio_mixer,
ringtone.path.normalized, volume=ringtone.volume, loop_count=0, pause_time=2)
            SIPApplication.voice_audio_bridge.add(self.wave_ringtone)
            self.wave_ringtone.start()
        ui.status = 'Ringing...'

    def _NH_SIPSessionWillStart(self, notification):
        ui = UI()
        if self.wave_ringtone:
            self.wave_ringtone.stop()
```

```
        ui.status = 'Connecting...'

        notification_center = NotificationCenter()
        notification_center.add_observer(self, sender=notification.sender.proposed_streams[0])

    def _NH_SIPSessionDidStart(self, notification):
        session = notification.sender

        ui = UI()
        ui.status = 'File transfer connected'

        identity = str(session.remote_identity.uri)
        if session.remote_identity.display_name:
            identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
        stream = session.streams[0]
        send_notice("File transfer for %s to '%s' started" % (stream.file_selector.name, identity))

    def _NH_SIPSessionDidFail(self, notification):
        notification_center = NotificationCenter()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)

        ui = UI()
        ui.status = None

        if self.wave_ringtone:
            self.wave_ringtone.stop()
        if notification.data.failure_reason == 'user request' and notification.data.code == 487:
            send_notice('File transfer cancelled')
        elif notification.data.failure_reason == 'user request':
            send_notice('File transfer rejected by user (%d %s)' % (notification.data.code,
notification.data.reason))
        else:
            send_notice('File transfer failed: %s' % notification.data.failure_reason)

    def _NH_SIPSessionDidEnd(self, notification):
        notification_center = NotificationCenter()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)
        notification_center.remove_observer(self, sender=session.streams[0] if session.streams else
session.proposed_streams[0])

        ui = UI()
        ui.status = None

        if not self.finished:
            send_notice('File transfer of %s canceled by %s party' % (os.path.basename(self.filepath),
notification.data.originator))

    def _NH_FileTransferStreamDidDeliverChunk(self, notification):
        ui = UI()
        ui.status = '%s: %s%%' % (os.path.basename(self.filepath),
notification.data.transferred_bytes*100//notification.data.file_size)

    def _NH_FileTransferStreamDidNotDeliverChunk(self, notification):
        send_notice('Failed to deliver chunk within file transfer of %s (%d %s)' %
(os.path.basename(self.filepath), notification.data.code, notification.data.reason))

    def _NH_FileTransferStreamDidFinish(self, notification):
        self.finished = True
        send_notice('File transfer of %s finished' % os.path.basename(self.filepath))
        self.session.end()


class IncomingTransferHandler(object):
    implements(IObserver)

    sessions = 0
    tone_ringtone = None

    def __init__(self, session, auto_answer_interval=None):
        self.session = session
        self.auto_answer_interval = auto_answer_interval
        self.file = None
        self.filename = None
        self.file_write_queue = EventQueue(self.write_chunk, name='File writing thread')
        self.finished = False
        self.hash = None
        self.question = None
        self.wave_ringtone = None

    def start(self):
```

```
        settings = SIPSimpleSettings()
        stream = self.session.proposed_streams[0]
        self.file_selector = stream.file_selector
        self.filename = filename = os.path.join(settings.file_transfer.directory.normalized,
self.file_selector.name)
        i = 1
        while os.path.exists(filename):
            filename = '%s.%d' % (self.filename, i)
            i += 1
        self.filename = filename
        try:
            self.file = open(self.filename, 'wb')
        except Exception, e:
            send_notice('Failed to open file "%s" for writing: %s' % (self.filename, e))
            self.session.reject(486)
            return
        self.hash = hashlib.sha1()

        IncomingTransferHandler.sessions += 1
        notification_center = NotificationCenter()
        notification_center.add_observer(self, sender=self.session)

        # start auto-answer
        self.answer_timer = None
        if self.auto_answer_interval == 0:
            self.session.accept(self.session.proposed_streams)
            return
        elif self.auto_answer_interval > 0:
            self.answer_timer = reactor.callFromThread(reactor.callLater, self.auto_answer_interval,
self.session.accept, self.session.proposed_streams)

        # start ringing
        application = SIPSessionApplication()
        if application.active_session is None:
            if IncomingTransferHandler.sessions == 1:
                ringtone = self.session.account.sounds.audio_inbound.sound_file if
self.session.account.sounds.audio_inbound is not None else None
                if ringtone:
                    self.wave_ringtone = WavePlayer(SIPApplication.alert_audio_mixer,
ringtone.path.normalized, volume=ringtone.volume, loop_count=0, pause_time=2)
                    SIPApplication.alert_audio_bridge.add(self.wave_ringtone)
                    self.wave_ringtone.start()
        elif IncomingTransferHandler.tone_ringtone is None:
            IncomingTransferHandler.tone_ringtone = WavePlayer(SIPApplication.voice_audio_mixer,
ResourcePath('sounds/ring_tone.wav').normalized, loop_count=0, pause_time=6)
            SIPApplication.voice_audio_bridge.add(IncomingTransferHandler.tone_ringtone)
            IncomingTransferHandler.tone_ringtone.start()
        self.session.send_ring_indication()

        # ask question
        identity = str(self.session.remote_identity.uri)
        if self.session.remote_identity.display_name:
            identity = '"%s" <%s>' % (self.session.remote_identity.display_name, identity)
        self.question = Question("Incoming file transfer for %s from '%s', do you want to accept?
(a)ccept/(r)eject" % (self.file_selector.name, identity), 'ari', bold=True)
        notification_center.add_observer(self, sender=self.question)
        ui = UI()
        ui.add_question(self.question)

    def write_chunk(self, data):
        if data is not None:
            self.file.write(data)
            self.hash.update(data)
        else:
            self.file.close()
            if self.finished:
                local_hash = 'sha1:' + ':'.join(re.findall(r'..', self.hash.hexdigest().upper()))
                remote_hash = self.file_selector.hash
                if local_hash != remote_hash:
                    send_notice('Warning: hash of transferred file does not match the remote hash
(file may have changed).')

    def handle_notification(self, notification):
        handler = getattr(self, '_NH_%s' % notification.name, Null())
        handler(notification)

    def _NH_UIQuestionGotAnswer(self, notification):
        notification_center = NotificationCenter()
        ui = UI()
        notification_center.remove_observer(self, sender=notification.sender)
        answer = notification.data.answer
        self.question = None
```

```
        if answer == 'a':
            self.session.accept(self.session.proposed_streams)
            ui.status = 'Accepting...'
        elif answer == 'r':
            self.session.reject()
            ui.status = 'Rejecting...'

        if IncomingTransferHandler.sessions == 1:
            if self.wave_ringtone:
                self.wave_ringtone.stop()
                self.wave_ringtone = None
            if IncomingTransferHandler.tone_ringtone:
                IncomingTransferHandler.tone_ringtone.stop()
                IncomingTransferHandler.tone_ringtone = None
        if self.answer_timer is not None and self.answer_timer.active():
            self.answer_timer.cancel()

    def _NH_SIPSessionWillStart(self, notification):
        ui = UI()
        if self.question is not None:
            notification_center = NotificationCenter()
            notification_center.remove_observer(self, sender=self.question)
            ui.remove_question(self.question)
            self.question = None
        ui.status = 'Connecting...'

        notification_center = NotificationCenter()
        notification_center.add_observer(self, sender=notification.sender.proposed_streams[0])

    def _NH_SIPSessionDidStart(self, notification):
        session = notification.sender
        IncomingCallInitializer.sessions -= 1

        ui = UI()
        ui.status = 'File transfer connected'

        identity = str(session.remote_identity.uri)
        if session.remote_identity.display_name:
            identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
        send_notice("File transfer for %s with '%s' started" % (self.file_selector.name, identity))

        self.file_write_queue.start()

        if IncomingTransferHandler.sessions == 1:
            if self.wave_ringtone:
                self.wave_ringtone.stop()
                self.wave_ringtone = None
            if IncomingTransferHandler.tone_ringtone:
                IncomingTransferHandler.tone_ringtone.stop()
                IncomingTransferHandler.tone_ringtone = None

    def _NH_SIPSessionDidFail(self, notification):
        notification_center = NotificationCenter()
        ui = UI()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)

        ui.status = None

        if self.question is not None:
            notification_center.remove_observer(self, sender=self.question)
            ui.remove_question(self.question)
            self.question = None

        IncomingTransferHandler.sessions -= 1
        if self.wave_ringtone:
            self.wave_ringtone.stop()
            self.wave_ringtone = None
        if IncomingTransferHandler.sessions == 0 and IncomingTransferHandler.tone_ringtone is not
None:
            IncomingTransferHandler.tone_ringtone.stop()
            IncomingTransferHandler.tone_ringtone = None
        if notification.data.failure_reason == 'user request' and notification.data.code == 487:
            send_notice('File transfer cancelled by user')
        elif notification.data.failure_reason == 'user request':
            send_notice('File transfer rejected (%d %s)' % (notification.data.code,
notification.data.reason))
        else:
            send_notice('File transfer failed: %s' % notification.data.failure_reason)

    def _NH_SIPSessionDidEnd(self, notification):
        notification_center = NotificationCenter()
```

```
        session = notification.sender
        notification_center.remove_observer(self, sender=session)
        notification_center.remove_observer(self, sender=session.streams[0] if session.streams else
session.proposed_streams[0])

        ui = UI()
        ui.status = None

        if not self.finished:
            send_notice('File transfer of %s canceled by %s party' %
(os.path.basename(self.file_selector.name), notification.data.originator))

        self.file_write_queue.put(None)
        self.file_write_queue.stop()

    def _NH_FileTransferStreamGotChunk(self, notification):
        ui = UI()
        ui.status = '%s: %s%%' % (os.path.basename(self.file_selector.name),
notification.data.transferred_bytes*100//notification.data.file_size)
        self.file_write_queue.put(notification.data.content)

    def _NH_FileTransferStreamDidFinish(self, notification):
        self.finished = True
        send_notice('File transfer of %s finished (file saved to "%s").' %
(os.path.basename(self.file_selector.name), self.filename))


class SIPSessionApplication(SIPApplication):
    # public methods
    #

    def __init__(self):
        self.account = None
        self.options = None
        self.target = None

        self.active_session = None
        self.outgoing_session = None
        self.connected_sessions = []
        self.sessions_with_proposals = set()
        self.hangup_timers = {}
        self.neighbours = set()
        self.registration_succeeded = False
        self.stopped_event = Event()

        self.ip_address_monitor = IPAddressMonitor()
        self.logger = None
        self.rtp_statistics = None
        self.nat_detector = None

        self.hold_tone = None

        self.ignore_local_hold = False
        self.ignore_local_unhold = False

    def start(self, target, options):
        notification_center = NotificationCenter()
        ui = UI()

        self.options = options
        self.target = target
        self.logger = Logger(sip_to_stdout=options.trace_sip, msrp_to_stdout=options.trace_msrp,
                             pjsip_to_stdout=options.trace_pjsip,
notifications_to_stdout=options.trace_notifications)

        notification_center.add_observer(self, sender=self)
        notification_center.add_observer(self, sender=ui)
        notification_center.add_observer(self, name='SIPSessionNewIncoming')
        notification_center.add_observer(self, name='SIPSessionNewOutgoing')
        notification_center.add_observer(self, name='AudioStreamDidChangeRTPParameters')
        notification_center.add_observer(self, name='AudioStreamICENegotiationDidSucceed')
        notification_center.add_observer(self, name='AudioStreamICENegotiationDidFail')

        log.level.current = log.level.WARNING # get rid of twisted messages
        control_bindings={'s': 'trace sip',
                          'm': 'trace msrp',
                          'j': 'trace pjsip',
                          'n': 'trace notifications',
                          'h': 'hangup',
                          'r': 'record',
                          'i': 'input',
                          'o': 'output',
```

```
                            'a': 'alert',
                            'u': 'mute',
                            ',': 'echo -',
                            '<': 'echo -',
                            '.': 'echo +',
                            '>': 'echo +',
                            ' ': 'hold',
                            'q': 'quit',
                            '/': 'help',
                            '?': 'help',
                            '0': 'dtmf 0',
                            '1': 'dtmf 1',
                            '2': 'dtmf 2',
                            '3': 'dtmf 3',
                            '4': 'dtmf 4',
                            '5': 'dtmf 5',
                            '6': 'dtmf 6',
                            '7': 'dtmf 7',
                            '8': 'dtmf 8',
                            '9': 'dtmf 9',
                            '*': 'dtmf *',
                            '#': 'dtmf #',
                            'A': 'dtmf A',
                            'B': 'dtmf B',
                            'C': 'dtmf C',
                            'D': 'dtmf D'}
        ui.start(control_bindings=control_bindings, display_text=False)

        Account.register_extension(AccountExtension)
        BonjourAccount.register_extension(AccountExtension)
        SIPSimpleSettings.register_extension(SIPSimpleSettingsExtension)
        try:
            SIPApplication.start(self, FileBackend(options.config_file or config_filename))
        except ConfigurationError, e:
            send_notice("Failed to load sipclient's configuration: %s\n" % str(e), bold=False)
            send_notice("If an old configuration file is in place, delete it or move it and recreate
the configuration using the sip_settings script.", bold=False)
            ui.stop()
            self.stopped_event.set()

    # notification handlers
    #

    def _NH_SIPApplicationWillStart(self, notification):
        account_manager = AccountManager()
        notification_center = NotificationCenter()
        settings = SIPSimpleSettings()
        ui = UI()

        for account in account_manager.iter_accounts():
            if isinstance(account, Account):
                account.sip.register = False
        if self.options.account is None:
            self.account = account_manager.default_account
        else:
            possible_accounts = [account for account in account_manager.iter_accounts() if
self.options.account in account.id and account.enabled]
            if len(possible_accounts) > 1:
                send_notice('More than one account exists which matches %s: %s' %
(self.options.account, ', '.join(sorted(account.id for account in possible_accounts))), bold=False)
                self.stop()
                return
            elif len(possible_accounts) == 0:
                send_notice('No enabled account which matches %s was found. Available and enabled
accounts: %s' % (self.options.account, ', '.join(sorted(account.id for account in
account_manager.get_accounts() if account.enabled))), bold=False)
                self.stop()
                return
            else:
                self.account = possible_accounts[0]
        notification_center.add_observer(self, sender=self.account)
        if isinstance(self.account, Account):
            self.account.sip.register = True
        send_notice('Using account %s' % self.account.id, bold=False)
        ui.prompt = Prompt(self.account.id, foreground='default')


        self.logger.start()
        if settings.logs.trace_sip and self.logger._siptrace_filename is not None:
            send_notice('Logging SIP trace to file "%s"' % self.logger._siptrace_filename, bold=False)
        if settings.logs.trace_msrp and self.logger._msrptrace_filename is not None:
```

```
            send_notice('Logging MSRP trace to file "%s"' % self.logger._msrptrace_filename,
bold=False)
        if settings.logs.trace_pjsip and self.logger._pjsiptrace_filename is not None:
            send_notice('Logging PJSIP trace to file "%s"' % self.logger._pjsiptrace_filename,
bold=False)
        if settings.logs.trace_notifications and self.logger._notifications_filename is not None:
            send_notice('Logging notifications trace to file "%s"' %
self.logger._notifications_filename, bold=False)

        if self.options.disable_sound:
            settings.audio.input_device = None
            settings.audio.output_device = None
            settings.audio.alert_device = None

        if isinstance(self.account, Account):
            self.nat_detector = NATDetector()

    def _NH_SIPApplicationDidStart(self, notification):
        settings = SIPSimpleSettings()

        self.ip_address_monitor.start()

        # set the file transfer directory if it's not set
        if settings.file_transfer.directory is None:
            settings.file_transfer.directory = 'file_transfers'

        # display a list of available devices
        self._CH_devices()

        send_notice('Type /help to see a list of available commands.', bold=False)

        if self.target is not None:
            call_initializer = OutgoingCallInitializer(self.account, self.target, audio=True)
            call_initializer.start()

    def _NH_SIPApplicationWillEnd(self, notification):
        self.ip_address_monitor.stop()

    def _NH_SIPApplicationDidEnd(self, notification):
        ui = UI()
        ui.stop()
        self.stopped_event.set()

    def _NH_UIInputGotCommand(self, notification):
        handler = getattr(self, '_CH_%s' % notification.data.command, None)
        if handler is not None:
            try:
                handler(*notification.data.args)
            except TypeError:
                send_notice('Illegal use of command /%s. Type /help for a list of available commands.'
% notification.data.command)
        else:
            send_notice('Unknown command /%s. Type /help for a list of available commands.' %
notification.data.command)

    def _NH_UIInputGotText(self, notification):
        msrp_chat = None
        if self.active_session is not None:
            try:
                msrp_chat = [stream for stream in self.active_session.streams if isinstance(stream,
ChatStream)][0]
            except IndexError:
                pass
        if msrp_chat is None:
            send_notice('No active chat session')
            return
        msrp_chat.send_message(notification.data.text)
        if msrp_chat.local_identity.display_name:
            local_identity = msrp_chat.local_identity.display_name
        else:
            local_identity = str(msrp_chat.local_identity.uri)
        ui = UI()
        ui.write(RichText('%s> ' % local_identity, foreground='darkred') + notification.data.text)

    def _NH_SIPEngineGotException(self, notification):
        lines = ['An exception occured within the SIP core:']
        lines.extend(notification.data.traceback.split('\n'))
        send_notice(lines)

    def _NH_SIPAccountRegistrationDidSucceed(self, notification):
        if self.registration_succeeded:
            return
```

```
        contact_header = notification.data.contact_header
        contact_header_list = notification.data.contact_header_list
        expires = notification.data.expires
        registrar = notification.data.registrar
        lines = ['%s Registered contact "%s" for sip:%s at %s:%d;transport=%s (expires in %d
seconds).' % (datetime.now().replace(microsecond=0), contact_header.uri, self.account.id,
registrar.address, registrar.port, registrar.transport, expires)]
        if len(contact_header_list) > 1:
            lines.append('Other registered contacts:')
            lines.extend('  %s (expires in %s seconds)' % (str(other_contact_header.uri),
other_contact_header.expires) for other_contact_header in contact_header_list if
other_contact_header.uri != notification.data.contact_header.uri)
        send_notice(lines)

        self.registration_succeeded = True

    def _NH_SIPAccountRegistrationDidFail(self, notification):
        send_notice('%s Failed to register contact for sip:%s: %s (retrying in %.2f seconds)' %
(datetime.now().replace(microsecond=0), self.account.id, notification.data.error,
notification.data.timeout))
        self.registration_succeeded = False

    def _NH_SIPAccountRegistrationDidEnd(self, notification):
        send_notice('%s Registration ended.' % datetime.now().replace(microsecond=0))

    def _NH_BonjourAccountRegistrationDidSucceed(self, notification):
        send_notice('%s Registered Bonjour contact "%s"' % (datetime.now().replace(microsecond=0),
notification.data.name))

    def _NH_BonjourAccountRegistrationDidFail(self, notification):
        send_notice('%s Failed to register Bonjour contact: %s' %
(datetime.now().replace(microsecond=0), notification.data.reason))

    def _NH_BonjourAccountRegistrationDidEnd(self, notification):
        send_notice('%s Registration ended.' % datetime.now().replace(microsecond=0))

    def _NH_BonjourAccountDidAddNeighbour(self, notification):
        if notification.data.uri not in self.neighbours:
            send_notice('%s Discovered Bonjour neighbour: "%s" <%s>' %
(datetime.now().replace(microsecond=0), notification.data.display_name, notification.data.uri))
            self.neighbours.add(BonjourNeighbour(notification.data.uri,
notification.data.display_name))

    def _NH_BonjourAccountDidRemoveNeighbour(self, notification):
        if notification.data.uri in self.neighbours:
            send_notice('%s Bonjour neighbour left: %s' % (datetime.now().replace(microsecond=0),
notification.data.uri))
            self.neighbours.remove(notification.data.uri)

    def _NH_BonjourAccountWillRestartDiscovery(self, notification):
        self.neighbours.clear()

    def _NH_SIPSessionNewIncoming(self, notification):
        session = notification.sender
        transfer_streams = [stream for stream in session.proposed_streams if stream.type == 'file-
transfer']
        # only allow sessions with 0 or 1 file transfers
        if len(transfer_streams) not in (0, 1):
            session.reject(488)
        if transfer_streams:
            transfer_handler = IncomingTransferHandler(session, self.options.auto_answer_interval)
            transfer_handler.start()
        else:
            notification_center = NotificationCenter()
            notification_center.add_observer(self, sender=session)
            call_initializer = IncomingCallInitializer(session, self.options.auto_answer_interval)
            call_initializer.start()

    def _NH_SIPSessionNewOutgoing(self, notification):
        session = notification.sender
        transfer_streams = [stream for stream in session.proposed_streams if stream.type == 'file-
transfer']
        if not transfer_streams:
            notification_center = NotificationCenter()
            notification_center.add_observer(self, sender=session)

    def _NH_SIPSessionDidFail(self, notification):
        notification_center = NotificationCenter()
        notification_center.remove_observer(self, sender=notification.sender)

    def _NH_SIPSessionWillStart(self, notification):
        notification_center = NotificationCenter()
```

```
        for stream in notification.sender.proposed_streams:
            notification_center.add_observer(self, sender=stream)

    def _NH_SIPSessionDidStart(self, notification):
        session = notification.sender

        self.connected_sessions.append(session)
        if self.active_session is not None:
            self.active_session.hold()
        self.active_session = session
        self._update_prompt()
        if len(self.connected_sessions) > 1:
            # this displays the conencted sessions
            self._CH_sessions()

        if self.options.auto_hangup_interval is not None:
            if self.options.auto_hangup_interval == 0:
                session.end()
            else:
                timer = reactor.callLater(self.options.auto_hangup_interval, session.end)
                self.hangup_timers[id(session)] = timer

    def _NH_SIPSessionWillEnd(self, notification):
        notification_center = NotificationCenter()
        session = notification.sender
        if id(session) in self.hangup_timers:
            timer = self.hangup_timers[id(session)]
            if timer.active():
                timer.cancel()
            del self.hangup_timers[id(session)]

        hangup_tone = WavePlayer(self.voice_audio_mixer,
ResourcePath('sounds/hangup_tone.wav').normalized)
        notification_center.add_observer(self, sender=hangup_tone)
        self.voice_audio_bridge.add(hangup_tone)
        hangup_tone.start()

    def _NH_SIPSessionDidEnd(self, notification):
        notification_center = NotificationCenter()
        session = notification.sender
        notification_center.remove_observer(self, sender=session)
        for stream in session.streams or session.proposed_streams:
            notification_center.remove_observer(self, sender=stream)

        ui = UI()
        ui.status = None

        identity = str(session.remote_identity.uri)
        if session.remote_identity.display_name:
            identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
        if notification.data.end_reason == 'user request':
            send_notice('SIP session with %s ended by %s party' % (identity,
notification.data.originator))
        else:
            send_notice('SIP session with %s ended due to error: %s' % (identity,
notification.data.end_reason))
        duration = session.end_time - session.start_time
        seconds = duration.seconds if duration.microseconds < 500000 else duration.seconds+1
        minutes, seconds = seconds / 60, seconds % 60
        hours, minutes = minutes / 60, minutes % 60
        hours += duration.days*24
        if not minutes and not hours:
            duration_text = '%d seconds' % seconds
        elif not hours:
            duration_text = '%02d:%02d' % (minutes, seconds)
        else:
            duration_text = '%02d:%02d:%02d' % (hours, minutes, seconds)
        send_notice('Session duration was %s' % duration_text)

        if session in self.connected_sessions:
            self.connected_sessions.remove(session)
        if session is self.active_session:
            if self.connected_sessions:
                self.active_session = self.connected_sessions[0]
                self.active_session.unhold()
                self.ignore_local_unhold = True
                identity = str(self.active_session.remote_identity.uri)
                if self.active_session.remote_identity.display_name:
                    identity = '"%s" <%s>' % (self.active_session.remote_identity.display_name,
identity)
                send_notice('Active SIP session: "%s" (%d/%d)' % (identity,
self.connected_sessions.index(self.active_session)+1, len(self.connected_sessions)))
```

```
            else:
                self.active_session = None
            self._update_prompt()

        on_hold_streams = [stream for stream in chain(*(session.streams for session in
self.connected_sessions)) if stream.on_hold]
        if not on_hold_streams and self.hold_tone:
            self.hold_tone.stop()

    def _NH_SIPSessionDidChangeHoldState(self, notification):
        session = notification.sender
        if notification.data.on_hold:
            if notification.data.originator == 'remote':
                if session is self.active_session:
                    send_notice('Remote party has put the session on hold')
                else:
                    identity = str(session.remote_identity.uri)
                    if session.remote_identity.display_name:
                        identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
                    send_notice('%s has put the session on hold' % identity)
            elif not self.ignore_local_hold:
                if session is self.active_session:
                    send_notice('Session is put on hold')
                else:
                    identity = str(session.remote_identity.uri)
                    if session.remote_identity.display_name:
                        identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
                    send_notice('Session with %s is put on hold' % identity)
            else:
                self.ignore_local_hold = False
        else:
            if notification.data.originator == 'remote':
                if session is self.active_session:
                    send_notice('Remote party has taken the session out of hold')
                else:
                    identity = str(session.remote_identity.uri)
                    if session.remote_identity.display_name:
                        identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
                    send_notice('%s has taken the session out of hold' % identity)
            elif not self.ignore_local_unhold:
                if session is self.active_session:
                    send_notice('Session is taken out of hold')
                else:
                    identity = str(session.remote_identity.uri)
                    if session.remote_identity.display_name:
                        identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
                    send_notice('Session with %s is taken out of hold' % identity)
            else:
                self.ignore_local_unhold = False

    def _NH_SIPSessionGotProposal(self, notification):
        self.sessions_with_proposals.add(notification.sender)
        if notification.data.originator == 'remote':
            proposal_handler = IncomingProposalHandler(notification.sender)
            proposal_handler.start()

    def _NH_SIPSessionDidRenegotiateStreams(self, notification):
        notification_center = NotificationCenter()
        for stream in notification.data.streams:
            if notification.data.action == 'add':
                notification_center.add_observer(self, sender=stream)
            elif notification.data.action == 'remove':
                notification_center.remove_observer(self, sender=stream)

        session = notification.sender
        streams = ', '.join(stream.type for stream in notification.data.streams)
        action = 'added' if notification.data.action == 'add' else 'removed'
        message = '%s party %s %s' % (notification.data.originator.capitalize(), action, streams)
        if session is not self.active_session:
            identity = str(session.remote_identity.uri)
            if session.remote_identity.display_name:
                identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
            message = '%s in session with %s' % (message, identity)
        send_notice(message)
        self._update_prompt()

    def _NH_AudioStreamGotDTMF(self, notification):
        notification_center = NotificationCenter()
        digit = notification.data.digit
        filename = 'sounds/dtmf_%s_tone.wav' % {'*': 'star', '#': 'pound'}.get(digit, digit)
        wave_player = WavePlayer(self.voice_audio_mixer, ResourcePath(filename).normalized)
        notification_center.add_observer(self, sender=wave_player)
```

```
        self.voice_audio_bridge.add(wave_player)
        wave_player.start()
        send_notice('Got DMTF %s' % notification.data.digit)

    def _NH_AudioStreamDidChangeHoldState(self, notification):
        if notification.data.on_hold:
            if not self.hold_tone:
                self.hold_tone = WavePlayer(self.voice_audio_mixer,
ResourcePath('sounds/hold_tone.wav').normalized, loop_count=0, pause_time=30, volume=50)
                self.voice_audio_bridge.add(self.hold_tone)
                self.hold_tone.start()
        else:
            on_hold_streams = [stream for stream in chain(*(session.streams for session in
self.connected_sessions)) if stream is not notification.sender and stream.on_hold]
            if not on_hold_streams and self.hold_tone:
                self.hold_tone.stop()
                self.hold_tone = None

    def _NH_AudioStreamDidChangeRTPParameters(self, notification):
        stream = notification.sender
        send_notice('Audio RTP parameters changed:')
        send_notice('Audio stream using "%s" codec at %sHz' % (stream.codec, stream.sample_rate))
        send_notice('Audio RTP endpoints %s:%d <-> %s:%d' % (stream.local_rtp_address,
stream.local_rtp_port, stream.remote_rtp_address, stream.remote_rtp_port))
        if stream.srtp_active:
            send_notice('RTP audio stream is encrypted')

    def _NH_AudioStreamDidStartRecordingAudio(self, notification):
        send_notice('Recording audio to %s' % notification.data.filename)

    def _NH_AudioStreamDidStopRecordingAudio(self, notification):
        send_notice('Stopped recording audio to %s' % notification.data.filename)

    def _NH_ChatStreamGotMessage(self, notification):
        if notification.data.message.sender.display_name:
            remote_identity = notification.data.message.sender.display_name
        else:
            remote_identity = notification.data.message.sender.uri
        ui = UI()
        ui.write(RichText('%s> ' % remote_identity, foreground='blue') +
notification.data.message.body)

    def _NH_DefaultAudioDeviceDidChange(self, notification):
        SIPApplication._NH_DefaultAudioDeviceDidChange(self, notification)
        if notification.data.changed_input and self.voice_audio_mixer.input_device=='system_default':
            send_notice('Switched default input device to: %s' %
self.voice_audio_mixer.real_input_device)
        if notification.data.changed_output and
self.voice_audio_mixer.output_device=='system_default':
            send_notice('Switched default output device to: %s' %
self.voice_audio_mixer.real_output_device)
        if notification.data.changed_output and
self.alert_audio_mixer.output_device=='system_default':
            send_notice('Switched alert device to: %s' % self.alert_audio_mixer.real_output_device)

    def _NH_AudioDevicesDidChange(self, notification):
        old_devices = set(notification.data.old_devices)
        new_devices = set(notification.data.new_devices)
        added_devices = new_devices - old_devices
        removed_devices = old_devices - new_devices
        changed_input_device = self.voice_audio_mixer.real_input_device in removed_devices
        changed_output_device = self.voice_audio_mixer.real_output_device in removed_devices
        changed_alert_device = self.alert_audio_mixer.real_output_device in removed_devices

        SIPApplication._NH_AudioDevicesDidChange(self, notification)

        if added_devices:
            send_notice('Added audio device(s): %s' % ', '.join(sorted(added_devices)))
        if removed_devices:
            send_notice('Removed audio device(s): %s' % ', '.join(sorted(removed_devices)))
        if changed_input_device:
            send_notice('Input device has been switched to: %s' %
self.voice_audio_mixer.real_input_device)
        if changed_output_device:
            send_notice('Output device has been switched to: %s' %
self.voice_audio_mixer.real_output_device)
        if changed_alert_device:
            send_notice('Alert device has been switched to: %s' %
self.alert_audio_mixer.real_output_device)

    def _NH_WavePlayerDidEnd(self, notification):
        notification_center = NotificationCenter()
```

```
        notification_center.remove_observer(self, sender=notification.sender)

    def _NH_AudioStreamICENegotiationDidSucceed(self, notification):
        send_notice("ICE negotiation succeeded in %s" % notification.data.duration)

    def _NH_AudioStreamICENegotiationDidFail(self, notification):
        stream = notification.sender
        send_notice("ICE negotiation failed: %s" % notification.data.reason)

    # command handlers
    #

    def _CH_call(self, target):
        if self.outgoing_session is not None:
            send_notice('Please cancel any outgoing sessions before makeing any new ones')
            return
        call_initializer = OutgoingCallInitializer(self.account, target, audio=True, chat=True)
        call_initializer.start()

    def _CH_audio(self, target, chat_option=None):
        if chat_option and chat_option != '+chat':
            raise TypeError()
        if self.outgoing_session is not None:
            send_notice('Please cancel any outgoing sessions before makeing any new ones')
            return
        call_initializer = OutgoingCallInitializer(self.account, target, audio=True,
chat=chat_option=='+chat')
        call_initializer.start()

    def _CH_chat(self, target, audio_option=None):
        if audio_option and audio_option != '+audio':
            raise TypeError()
        if self.outgoing_session is not None:
            send_notice('Please cancel any outgoing sessions before makeing any new ones')
            return
        call_initializer = OutgoingCallInitializer(self.account, target, audio=audio_option=='+audio',
chat=True)
        call_initializer.start()

    def _CH_send(self, target, filepath):
        transfer_handler = OutgoingTransferHandler(self.account, target, filepath)
        transfer_handler.start()

    def _CH_next(self):
        if len(self.connected_sessions) > 1:
            self.active_session.hold()
            self.active_session =
self.connected_sessions[(self.connected_sessions.index(self.active_session)+1) %
len(self.connected_sessions)]
            self.active_session.unhold()
            self.ignore_local_unhold = True
            identity = str(self.active_session.remote_identity.uri)
            if self.active_session.remote_identity.display_name:
                identity = '"%s" <%s>' % (self.active_session.remote_identity.display_name, identity)
            send_notice('Active SIP session: "%s" (%d/%d)' % (identity,
self.connected_sessions.index(self.active_session)+1, len(self.connected_sessions)))
            self._update_prompt()

    def _CH_prev(self):
        if len(self.connected_sessions) > 1:
            self.active_session.hold()
            self.active_session =
self.connected_sessions[self.connected_sessions.index(self.active_session)-1]
            self.active_session.unhold()
            self.ignore_local_unhold = True
            identity = str(self.active_session.remote_identity.uri)
            if self.active_session.remote_identity.display_name:
                identity = '"%s" <%s>' % (self.active_session.remote_identity.display_name, identity)
            send_notice('Active SIP session: "%s" (%d/%d)' % (identity,
self.connected_sessions.index(self.active_session)+1, len(self.connected_sessions)))
            self._update_prompt()

    def _CH_sessions(self):
        if self.connected_sessions:
            lines = ['Connected sessions:']
            for session in self.connected_sessions:
                identity = str(session.remote_identity.uri)
                if session.remote_identity.display_name:
                    identity = '"%s" <%s>' % (session.remote_identity.display_name, identity)
                lines.append('  SIP session with %s (%d/%d) - %s' % (identity,
self.connected_sessions.index(session)+1, len(self.connected_sessions), 'active' if session is
self.active_session else 'on hold'))
```

```
            if len(self.connected_sessions) > 1:
                lines.append('Use the /next and /prev commands to switch the active session')
            send_notice(lines)
        else:
            send_notice('There are no connected sessions')

    def _CH_neighbours(self):
        if not isinstance(self.account, BonjourAccount):
            send_notice('This command is only available if using the Bonjour account')
            return
        lines = ['Bonjour neighbours:']
        for neighbour in sorted(self.neighbours, key=attrgetter('display_name', 'uri')):
            if neighbour.display_name:
                lines.append('  "%s" <%s>' % (neighbour.display_name, neighbour.uri))
            else:
                lines.append('  %s' % neighbour.uri)
        send_notice(lines)

    def _CH_trace(self, *types):
        if not types:
            lines = []
            lines.append('SIP tracing to console is now %s' % ('active' if self.logger.sip_to_stdout
else 'inactive'))
            lines.append('MSRP tracing to console is now %s' % ('active' if self.logger.msrp_to_stdout
else 'inactive'))
            lines.append('PJSIP tracing to console is now %s' % ('active' if
self.logger.pjsip_to_stdout else 'inactive'))
            lines.append('Notification tracing to console is now %s' % ('active' if
self.logger.notifications_to_stdout else 'inactive'))
            send_notice(lines)
            return

        add_types = [type[1:] for type in types if type[0] == '+']
        remove_types = [type[1:] for type in types if type[0] == '-']
        toggle_types = [type for type in types if type[0] not in ('+', '-')]

        if 'sip' in add_types or ('sip' in toggle_types and not self.logger.sip_to_stdout):
            self.logger.sip_to_stdout = True
            send_notice('SIP tracing to console is now activated')
        elif 'sip' in remove_types or ('sip' in toggle_types and self.logger.sip_to_stdout):
            self.logger.sip_to_stdout = False
            send_notice('SIP tracing to console is now deactivated')

        if 'msrp' in add_types or ('msrp' in toggle_types and not self.logger.msrp_to_stdout):
            self.logger.msrp_to_stdout = True
            send_notice('MSRP tracing to console is now activated')
        elif 'msrp' in remove_types or ('msrp' in toggle_types and self.logger.msrp_to_stdout):
            self.logger.msrp_to_stdout = False
            send_notice('MSRP tracing to console is now deactivated')

        if 'pjsip' in add_types or ('pjsip' in toggle_types and not self.logger.pjsip_to_stdout):
            self.logger.pjsip_to_stdout = True
            send_notice('PJSIP tracing to console is now activated')
        elif 'pjsip' in remove_types or ('pjsip' in toggle_types and self.logger.pjsip_to_stdout):
            self.logger.pjsip_to_stdout = False
            send_notice('PJSIP tracing to console is now deactivated')

        if 'notifications' in add_types or ('notifications' in toggle_types and not
self.logger.notifications_to_stdout):
            self.logger.notifications_to_stdout = True
            send_notice('Notification tracing to console is now activated')
        elif 'notifications' in remove_types or ('notifications' in toggle_types and
self.logger.notifications_to_stdout):
            self.logger.notifications_to_stdout = False
            send_notice('Notification tracing to console is now deactivated')

    def _CH_rtp(self, state='toggle'):
        if state == 'toggle':
            new_state = self.rtp_statistics is None
        elif state == 'on':
            new_state = True
        elif state == 'off':
            new_state = False
        else:
            raise TypeError()
        if (self.rtp_statistics and new_state) or (not self.rtp_statistics and not new_state):
            return
        if new_state:
            self.rtp_statistics = RTPStatisticsThread()
            self.rtp_statistics.start()
            send_notice('Output of RTP statistics on console is now activated')
        else:
```

```
            self.rtp_statistics.stop()
            self.rtp_statistics = None
            send_notice('Output of RTP statistics on console is now dectivated')

    def _CH_mute(self, state='toggle'):
        if state == 'toggle':
            self.voice_audio_mixer.muted = not self.voice_audio_mixer.muted
        elif state == 'on':
            self.voice_audio_mixer.muted = True
        elif state == 'off':
            self.voice_audio_mixer.muted = False
        send_notice('The microphone is now %s' % ('muted' if self.voice_audio_mixer.muted else
'unmuted'))

    def _CH_input(self, device=None):
        engine = Engine()
        input_devices = [None, 'system_default'] + sorted(engine.input_devices)
        if device is None:
            if self.voice_audio_mixer.input_device in input_devices:
                old_input_device = self.voice_audio_mixer.input_device
            else:
                old_input_device = None
            tries = 0
            while tries < len(input_devices):
                new_input_device = input_devices[(input_devices.index(old_input_device)+1) %
len(input_devices)]
                try:
                    self.voice_audio_mixer.set_sound_devices(new_input_device,
self.voice_audio_mixer.output_device, self.voice_audio_mixer.ec_tail_length)
                except SIPCoreError, e:
                    tries += 1
                    old_input_device = new_input_device
                    send_notice('Failed to set input device to %s: %s' % (new_input_device, str(e)))
                else:
                    if new_input_device == 'system_default':
                        send_notice('Input device changed to %s (system default device)' %
self.voice_audio_mixer.real_input_device)
                    else:
                        send_notice('Input device changed to %s' % new_input_device)
                    break
        else:
            if device == 'None':
                device = None
            elif device not in input_devices:
                send_notice('Unknown input device %s. Type /devices to see a list of available
devices' % device)
                return
            try:
                self.voice_audio_mixer.set_sound_devices(device, self.voice_audio_mixer.output_device,
self.voice_audio_mixer.ec_tail_length)
            except SIPCoreError, e:
                send_notice('Failed to set input device to %s: %s' % (device, str(e)))
            else:
                if device == 'system_default':
                    send_notice('Input device changed to %s (system default device)' %
self.voice_audio_mixer.real_input_device)
                else:
                    send_notice('Input device changed to %s' % device)

    def _CH_output(self, device=None):
        engine = Engine()
        output_devices = [None, 'system_default'] + sorted(engine.output_devices)
        if device is None:
            if self.voice_audio_mixer.output_device in output_devices:
                old_output_device = self.voice_audio_mixer.output_device
            else:
                old_output_device = None
            tries = 0
            while tries < len(output_devices):
                new_output_device = output_devices[(output_devices.index(old_output_device)+1) %
len(output_devices)]
                try:
                    self.voice_audio_mixer.set_sound_devices(self.voice_audio_mixer.input_device,
new_output_device, self.voice_audio_mixer.ec_tail_length)
                except SIPCoreError, e:
                    tries += 1
                    old_output_device = new_output_device
                    send_notice('Failed to set output device to %s: %s' % (new_output_device, str(e)))
                else:
                    if new_output_device == 'system_default':
                        send_notice('Output device changed to %s (system default device)' %
self.voice_audio_mixer.real_output_device)
```

```
                    else:
                        send_notice('Output device changed to %s' % new_output_device)
                    break
        else:
            if device == 'None':
                device = None
            elif device not in output_devices:
                send_notice('Unknown output device %s. Type /devices to see a list of available
devices' % device)
                return
            try:
                self.voice_audio_mixer.set_sound_devices(self.voice_audio_mixer.input_device, device,
self.voice_audio_mixer.ec_tail_length)
            except SIPCoreError, e:
                send_notice('Failed to set output device to %s: %s' % (device, str(e)))
            else:
                if device == 'system_default':
                    send_notice('Output device changed to %s (system default device)' %
self.voice_audio_mixer.real_output_device)
                else:
                    send_notice('Output device changed to %s' % device)

    def _CH_alert(self, device=None):
        engine = Engine()
        output_devices = [None, 'system_default'] + sorted(engine.output_devices)
        if device is None:
            if self.alert_audio_mixer.output_device in output_devices:
                old_output_device = self.alert_audio_mixer.output_device
            else:
                old_output_device = None
            tries = 0
            while tries < len(output_devices):
                new_output_device = output_devices[(output_devices.index(old_output_device)+1) %
len(output_devices)]
                try:
                    self.alert_audio_mixer.set_sound_devices(self.alert_audio_mixer.input_device,
new_output_device, self.alert_audio_mixer.ec_tail_length)
                except SIPCoreError, e:
                    tries += 1
                    old_output_device = new_output_device
                    send_notice('Failed to set alert device to %s: %s' % (new_output_device, str(e)))
                else:
                    if new_output_device == 'system_default':
                        send_notice('Alert device changed to %s (system default device)' %
self.alert_audio_mixer.real_output_device)
                    else:
                        send_notice('Alert device changed to %s' % new_output_device)
                    break
        else:
            if device == 'None':
                device = None
            elif device not in output_devices:
                send_notice('Unknown output device %s. Type /devices to see a list of available
devices' % device)
                return
            try:
                self.alert_audio_mixer.set_sound_devices(self.alert_audio_mixer.input_device, device,
self.alert_audio_mixer.ec_tail_length)
            except SIPCoreError, e:
                send_notice('Failed to set alert device to %s: %s' % (device, str(e)))
            else:
                if device == 'system_default':
                    send_notice('Alert device changed to %s (system default device)' %
self.alert_audio_mixer.real_output_device)
                else:
                    send_notice('Alert device changed to %s' % device)

    def _CH_devices(self):
        engine = Engine()
        send_notice('Available audio input devices: %s' % ', '.join(['None', 'system_default'] +
sorted(engine.input_devices)), bold=False)
        send_notice('Available audio output devices: %s' % ', '.join(['None', 'system_default'] +
sorted(engine.output_devices)), bold=False)
        if self.voice_audio_mixer.input_device == 'system_default':
            send_notice('Using audio input device: %s (system default device)' %
self.voice_audio_mixer.real_input_device, bold=False)
        else:
            send_notice('Using audio input device: %s' % self.voice_audio_mixer.input_device,
bold=False)
        if self.voice_audio_mixer.output_device == 'system_default':
            send_notice('Using audio output device: %s (system default device)' %
self.voice_audio_mixer.real_output_device, bold=False)
```

```
        else:
            send_notice('Using audio output device: %s' % self.voice_audio_mixer.output_device,
bold=False)
        if self.alert_audio_mixer.output_device == 'system_default':
            send_notice('Using audio alert device: %s (system default device)' %
self.alert_audio_mixer.real_output_device, bold=False)
        else:
            send_notice('Using audio alert device: %s' % self.alert_audio_mixer.output_device,
bold=False)

    def _CH_echo(self, adjust=None):
        if adjust is None:
            send_notice('Echo cancellation tail length is %d ms' %
self.voice_audio_mixer.ec_tail_length)
            return
        adjust_match = re.match(r'(?P<sign>\+|\-)?(?P<value>[0-9]+)', adjust)
        if adjust_match is None:
            raise TypeError()
        sign, value = adjust_match.groups()
        value = int(value)
        if sign is None:
            new_tail_length = value
        elif sign == '+':
            new_tail_length = self.voice_audio_mixer.ec_tail_length + value
        elif sign == '-':
            new_tail_length = self.voice_audio_mixer.ec_tail_length - value
        if new_tail_length < 0:
            new_tail_length = 0
        if new_tail_length > 500:
            new_tail_length = 500
        if new_tail_length != self.voice_audio_mixer.ec_tail_length:
            self.voice_audio_mixer.set_sound_devices(self.voice_audio_mixer.input_device,
self.voice_audio_mixer.output_device, new_tail_length)
        send_notice('Set the echo cancellation tail length to %d ms' %
self.voice_audio_mixer.ec_tail_length)

    def _CH_help(self):
        self._print_help()

    def _CH_quit(self):
        self.stop()

    def _CH_eof(self):
        ui = UI()
        if self.active_session is not None:
            if self.active_session in self.sessions_with_proposals:
                ui.status = 'Cancelling proposal...'
                self.active_session.cancel_proposal()
            else:
                ui.status = 'Ending SIP session...'
                self.active_session.end()
        elif self.outgoing_session is not None:
            ui.status = 'Cancelling SIP session...'
            self.outgoing_session.end()
        else:
            self.stop()

    def _CH_hangup(self):
        if self.active_session is not None:
            send_notice('Ending SIP session...')
            self.active_session.end()
        elif self.outgoing_session is not None:
            send_notice('Cancelling SIP session...')
            self.outgoing_session.end()

    @run_in_green_thread
    def _CH_dtmf(self, tones):
        if self.active_session is not None:
            try:
                audio_stream = [stream for stream in self.active_session.streams if isinstance(stream,
AudioStream)][0]
            except IndexError:
                pass
            else:
                notification_center = NotificationCenter()
                for digit in tones:
                    audio_stream.send_dtmf(digit)
                    filename = 'sounds/dtmf_%s_tone.wav' % {'*': 'star', '#': 'pound'}.get(digit,
digit)
                    wave_player = WavePlayer(self.voice_audio_mixer,
ResourcePath(filename).normalized)
                    notification_center.add_observer(self, sender=wave_player)
```

```
                    if self.active_session.account.rtp.inband_dtmf:
                        audio_stream.bridge.add(wave_player)
                    else:
                        self.voice_audio_bridge.add(wave_player)
                    wave_player.start()
                    api.sleep(0.3)

    def _CH_record(self, state='toggle'):
        if self.active_session is None:
            return
        try:
            audio_stream = [stream for stream in self.active_session.streams if isinstance(stream,
AudioStream)][0]
        except IndexError:
            pass
        else:
            if state == 'toggle':
                new_state = not audio_stream.recording_active
            elif state == 'on':
                new_state = True
            elif state == 'off':
                new_state = False
            else:
                send_notice('Illegal argument to /record. Type /help for a list of available
commands.')
                return
            if new_state:
                audio_stream.start_recording()
            else:
                audio_stream.stop_recording()

    def _CH_hold(self, state='toggle'):
        if self.active_session is not None:
            if state == 'toggle':
                new_state = not self.active_session.on_hold
            elif state == 'on':
                new_state = True
            elif state == 'off':
                new_state = False
            else:
                send_notice('Illegal argument to /hold. Type /help for a list of available commands.')
                return
            if new_state:
                self.active_session.hold()
            else:
                self.active_session.unhold()

    def _CH_add(self, stream_name):
        if self.active_session is None:
            send_notice('There is no active session')
            return
        if stream_name in (stream.type for stream in self.active_session.streams):
            send_notice('The active session already has a %s stream' % stream_name)
            return
        proposal_handler = OutgoingProposalHandler(self.active_session, **{stream_name: True})
        try:
            proposal_handler.start()
        except IllegalStateError:
            send_notice('Cannot add a stream while another transaction is in progress')

    def _CH_remove(self, stream_name):
        if self.active_session is None:
            send_notice('There is no active session')
            return
        try:
            stream = (stream for stream in self.active_session.streams if
stream.type==stream_name).next()
        except StopIteration:
            send_notice('The current active session does not have any %s streams' % stream_name)
        else:
            try:
                self.active_session.remove_stream(stream)
            except IllegalStateError:
                send_notice('Cannot remove a stream while another transaction is in progress')

    # private methods
    #

    def _print_help(self):
        lines = []
        lines.append('General commands:')
        lines.append('  /call {user[@domain]}: call the specified user using audio and chat')
```

```
        lines.append('  /audio {user[@domain]} [+chat]: call the specified user using audio and
possibly chat')
        lines.append('  /chat {user[@domain]} [+audio]: call the specified user using chat and
possibly audio')
        lines.append('  /send {user[@domain]} {file}: initiate a file transfer with the specified
user')
        lines.append('  /next: select the next connected session')
        lines.append('  /prev: select the previous connected session')
        lines.append('  /sessions: show the list of connected sessions')
        if isinstance(self.account, BonjourAccount):
            lines.append('  /neighbours: show the list of bonjour neighbours')
        lines.append('  /trace [[+|-]sip] [[+|-]msrp] [[+|-]pjsip] [[+|-]notifications]: toggle/set
tracing on the console (ctrl-x s | ctrl-x m | ctrl-x j | ctrl-x n)')
        lines.append('  /rtp [on|off]: toggle/set printing RTP statistics on the console (ctrl-x p)')
        lines.append('  /mute [on|off]: mute the microphone (ctrl-x u)')
        lines.append('  /input [device]: change audio input device (ctrl-x i)')
        lines.append('  /output [device]: change audio output device (ctrl-x o)')
        lines.append('  /alert [device]: change audio alert device (ctrl-x a)')
        lines.append('  /echo [+|-][value]: adjust echo cancellation (ctrl-x < | ctrl-x >)')
        lines.append('  /quit: quit the program (ctrl-x q)')
        lines.append('  /help: display this help message (ctrl-x ?)')
        lines.append('In call commands:')
        lines.append('  /hangup: hang-up the active session (ctrl-x h)')
        lines.append('  /dtmf {0-9|*|#|A-D}...: send DTMF tones (ctrl-x 0-9|*|#|A-D)')
        lines.append('  /record [on|off]: toggle/set audio recording (ctrl-x r)')
        lines.append('  /hold [on|off]: hold/unhold (ctrl-x SPACE)')
        lines.append('  /add {chat|audio}: add a stream to the current session')
        lines.append('  /remove {chat|audio}: remove a stream from the current session')
        send_notice(lines, bold=False)

    def _update_prompt(self):
        ui = UI()
        session = self.active_session
        if session is None:
            ui.prompt = Prompt(self.account.id, foreground='default')
        else:
            identity = '%s@%s' % (session.remote_identity.uri.user, session.remote_identity.uri.host)
            if session.remote_identity.display_name:
                identity = '%s (%s)' % (session.remote_identity.display_name, identity)
            streams = '/'.join(stream.type.capitalize() for stream in session.streams)
            if not streams:
                streams = 'Session without media'
            ui.prompt = Prompt('%s to %s' % (streams, identity), foreground='darkred')


def parse_handle_call_option(option, opt_str, value, parser, name):
    try:
        value = parser.rargs[0]
    except IndexError:
        value = 0
    else:
        if value == '' or value[0] == '-':
            value = 0
        else:
            try:
                value = int(value)
            except ValueError:
                value = 0
            else:
                del parser.rargs[0]
    setattr(parser.values, name, value)

if __name__ == '__main__':
    description = '%prog is a command-line client for handling multiple audio, chat and file-transfer
sessions'
    usage = '%prog [options] [user@domain]'
    parser = OptionParser(usage=usage, description=description)
    parser.print_usage = parser.print_help
    parser.add_option('-a', '--account', type='string', dest='account', help='The account name to use
for any outgoing traffic. If not supplied, the default account will be used.', metavar='NAME')
    parser.add_option('-c', '--config-file', type='string', dest='config_file', help='The path to a
configuration file to use. This overrides the default location of the configuration file.',
metavar='FILE')
    parser.add_option('-s', '--trace-sip', action='store_true', dest='trace_sip', default=False,
help='Dump the raw contents of incoming and outgoing SIP messages.')
    parser.add_option('-m', '--trace-msrp', action='store_true', dest='trace_msrp', default=False,
help='Dump msrp logging information and the raw contents of incoming and outgoing MSRP messages.')
    parser.add_option('-j', '--trace-pjsip', action='store_true', dest='trace_pjsip', default=False,
help='Print PJSIP logging output.')
    parser.add_option('-n', '--trace-notifications', action='store_true', dest='trace_notifications',
default=False, help='Print all notifications (disabled by default).')
```

```
    parser.add_option('-S', '--disable-sound', action='store_true', dest='disable_sound',
default=False, help='Disables initializing the sound card.')
    parser.set_default('auto_answer_interval', None)
    parser.add_option('--auto-answer', action='callback', callback=parse_handle_call_option,
callback_args=('auto_answer_interval',), help='Interval after which to answer an incoming session
(disabled by default). If the option is specified but the interval is not, it defaults to 0 (accept
the session as soon as it starts ringing).', metavar='[INTERVAL]')
    parser.set_default('auto_hangup_interval', None)
    parser.add_option('--auto-hangup', action='callback', callback=parse_handle_call_option,
callback_args=('auto_hangup_interval',), help='Interval after which to hang up an established session
(disabled by default). If the option is specified but the interval is not, it defaults to 0 (hangup
the session as soon as it connects).', metavar='[INTERVAL]')
    options, args = parser.parse_args()

    target = args[0] if args else None

    application = SIPSessionApplication()
    application.start(target, options)

    signal.signal(signal.SIGINT, signal.SIG_DFL)
    application.stopped_event.wait()
    sleep(0.1)
```