switchy Documentation

Release 0.1.alpha

Tyler Goodlet

August 30, 2016

Fast FreeSWITCH control purpose-built upon traffic theory and stress testing.

CHAPTER 1

Overview

Switchy intends to be a *fast* control library for harnessing the power of the *FreeSWITCH* telephony engine whilst leveraging the expressiveness of Python. It relies on the *FreeSWITCH* ESL inbound method for control and was originally built for generating traffic using *FreeSWITCH* clusters.

CHAPTER 2

Installation and Dependencies

See instructions on the github page.

CHAPTER 3

Features

- drive multiple *FreeSWITCH* processes as a traffic generator
- write services in pure Python to process flows from a *FreeSWITCH* cluster
- build a dialplan system using a *Flask-like routing* API
- record, display and export CDR and performance metrics captured during stress tests
- async without requiring `twisted`

3.1 *FreeSWITCH* Configuration

Switchy relies on some baseline server *deployment* steps for import-and-go usage.

CHAPTER 4

User Guide

4.1 *FreeSWITCH* configuration and deployment

switchy relies on some basic *FreeSWITCH* configuration steps in order to enable remote control via the ESL inbound method. Most importantly, the ESL configuration file must be modified to listen on a known socket of choice and a *park-only* extension must be added to *FreeSWITCH*'s XML dialplan. *switchy* comes packaged with an example *park only dialplan* which you can copy-paste into your existing server(s).

4.1.1 Event Socket

In order for *switchy* to talk to *FreeSWITCH* you must enable ESL to listen on all IP addrs at port *8021*. This can configured by simply making the following change to the `${FS_CONF_ROOT}/conf/autoload_configs/event_socket.conf.xml` configuration file:

```
-- <param name="listen-ip" value="127.0.0.1"/>
++ <param name="listen-ip" value="::"/>
```

Depending on your FS version, additional acl configuration may be required.

4.1.2 Park only dialplan

An XML dialplan extension which places all *inbound* sessions into the park state should be added to all target *FreeSWITCH* servers you wish to control with *switchy*. An example context (`switchydp.xml`) is included in the conf directory of the source code. If using this file you can enable *switchy* to control all calls received by a particular *FreeSWITCH* SIP profile by setting the `"switchy"` context.

As an example you can modify *FreeSWITCH*'s default external profile found at `${FS_CONF_ROOT}/conf/sip_profiles/external.xml`:

```
<!-- Contents of  -->
-- <param name="context" value="public"/>
++ <param name="context" value="switchy"/>
```

Note: You can also add a park extension to your existing dialplan such that only a subset of calls relinquish control to *switchy* (especially useful if you'd like to test on an extant production system).

4.1.3 Configuring software under test

For (stress) testing, the system under test should be configured to route calls back to the originating *FreeSWITCH* (cluster) such that the originator hosts both the *caller* and *callee* user agents (potentially using the same SIP profile):

```
FreeSWITCH cluster                    Target test network or
                                      device

--------------    outbound sessions   ---------------------
| Originator | ---------------------> | Device under test |
|            | <--------------------- |   (in loopback)   |
--------------    inbound sessions    ---------------------
```

This allows *switchy* to perform *call tracking* (associate *outbound* with *inbound* SIP sessions) and thus assume full control of call flow as well as measure signalling latency and other teletraffic metrics.

Example *proxy* dialplan

If your system to test is simply another *FreeSWITCH* instance then it is highly recommended to use a *"proxy"* dialplan to route SIP sessions back to the originator (caller):

```
<!-- Proxy Dialplan - forward calls to requested destination -->
<condition field="${sip_req_uri}" expression="^(.+)$">
    <action application="bridge" data="sofia/${sofia_profile_name}/${sip_req_uri}"/>
</condition>
```

Note: This could have alternatively be implemented as a *switchy* app.

Configuring FreeSWITCH for stress testing

Before attempting to stress test *FreeSWITCH* itself be sure you've read the performance and dialplans sections of the wiki.

You'll typically want to raise the *max-sessions* and *sessions-per-second* parameters in *autoload_configs/switch.conf.xml*:

```
<param name="max-sessions" value="20000"/>
<!-- Max channels to create per second -->
<param name="sessions-per-second" value="1000"/>
```

This prevents *FreeSWITCH* from rejecting calls at high loads. However, if your intention is to see how *FreeSWITCH* behaves at those parameters limits, you can always set values that suit those purposes.

In order to reduce load due to logging it's recommended you reduce your core logging level. This is also done in *autoload_configs/switch.conf.xml*:

```
<!-- Default Global Log Level - value is one of debug,info,notice,warning,err,crit,alert -->
<param name="loglevel" value="warning"/>
```

You will also probably want to raise the file descriptor count.

Note: You have to run *ulimit* in the same shell where you start a *FreeSWITCH* process.

4.2 Connection wrappers

ESL connection wrapper

class `switchy.connection.`**`Connection`**(*host*, *port='8021'*, *auth='ClueCon'*, *locked=True*, *lock=None*)

Connection wrapper which can provide mutex attr access making the underlying ESL.ESLconnection thread safe.

(Note: must be explicitly connected before use.)

`api`(*cmd*, *errcheck=True*)

Invoke esl api command (with error checking by default). Returns an ESL.ESLEvent instance for event type "SOCKET_DATA".

`cmd`(*cmd*)

Return the string-body output from invoking a command.

`connect`(*host=None*, *port=None*, *auth=None*)

Reconnect if disconnected

`connected`()

Return bool indicating if this connection is active

`disconnect`()

Rewrap disconnect to avoid deadlocks

`subscribe`(*event_types*, *fmt='plain'*)

Subscribe connection to receive events for all names in *event_types*

`switchy.connection.`**`check_con`**(*con*)

Raise a connection error if this connection is down.

4.3 Observer components

Observer machinery.

Includes components for observing and controlling FreeSWITCH server state through event processing and command invocation.

class `switchy.observe.`**`Client`**(*host='127.0.0.1'*, *port='8021'*, *auth='ClueCon'*, *call_tracking_header=None*, *listener=None*, *logger=None*)

Interface for synchronous server control using the esl "inbound method" as described here: https://wiki.freeswitch.org/wiki/Mod_event_socket#Inbound

Provides a high level interface for interaction with an event listener.

`api`(*cmd*, *exc=True*)

Invoke esl api command with error checking Returns an ESL.ESLEvent instance for event type "SOCKET_DATA".

`bgapi`(*cmd*, *listener=None*, *callback=None*, *client_id=None*, ***jobkwargs*)

Execute a non blocking api call and handle it to completion

cmd [string] command to execute

listener [EvenListener instance] listener which will handle bg job events for this cmd

callback [callable] Object to call once the listener collects the bj event result. By default the listener calls back the job instance with the response from the 'BACKGROUND_JOB' event's body content plus any kwargs passed here.

cmd(*cmd*)
Return the string-body output from invoking a command

connect()
Connect this client

connected()
Check if connection is active

disconnect()
Disconnect the client's underlying connection

hupall(*group_id=None*)
Hangup all calls associated with this client by iterating all managed call apps and hupall-ing with the apps callback id. If **:var:'group_id'** is provided look up the corresponding app an hang up calls for that specific app.

load_app(*ns*, *on_value=None*, *header=None*, *prepend=False*, *funcargsmap=None*, ***prepost_kwargs*)
Load annotated callbacks and from a namespace and add them to this client's listener's callback chain.

Parameters **ns** – A namespace-like object containing functions marked with @event_callback (can be a module, class or instance).

Params str on_value app group id key to be used for registering app callbacks with the *EventListener*. This value will be inserted in the *originate* command as an X-header and used to look up which app callbacks should be invoked for each received event.

originate(*dest_url=None*, *uuid_func=<function uuid>*, *app_id=None*, *listener=None*, *bgapi_kwargs={}*, *rep_fields={}*, ***orig_kwargs*)
Originate a call using FreeSWITCH 'originate' command. A non-blocking bgapi call is used by default.

see `build_originate_cmd()`

orig_kwargs: additional originate cmd builder kwargs forwarded to `build_originate_cmd()` call

instance of *Job* a background job

set_orig_cmd(**args*, ***kwargs*)
Return a formatted *originate* command string conforming to the syntax dictated by mod_commands of the form:

originate <call url> <exten>|&<application_name>(<app_args>) [<dialplan>] [<context>] [<cid_name>] [<cid_num>] [<timeout_sec>]

dest_url [str] call destination url with format <username_uri>@<domain>:<port>

profile [str] sofia profile (UA) name to use for making outbound call

dp_extension: str destination dp extension where the originating session (a-leg) will processed just after the call is answered

etc...

originate command [string or callable] full cmd string if uuid_str is not None, else callable f(uuid_str) -> full cmd string

unload_app(*on_value*, *ns=None*)
Unload all callbacks associated with a particular app *on_value* id. If *ns* is provided unload only the callbacks from that particular subapp.

class `switchy.observe.`**`EventListener`**(*host='127.0.0.1', port='8021', auth='ClueCon', session_map=None, bg_jobs=None, rx_con=None, call_tracking_header='variable_call_uuid', app_id_headers=None, autorecon=30, max_limit=inf, _tx_lock=None*)

ESL Listener which tracks FreeSWITCH state using an observer pattern. This implementation utilizes a background event loop (single thread) and one *Connection*.

The main purpose is to enable event oriented state tracking of various slave process objects and call entities.

`add_callback`(*evname, ident, callback, *args, **kwargs*)

Register a callback for events of type *evname* to be called with provided args, kwargs when an event is received by this listener.

evname [string] name of mod_event event type you wish to subscribe for with the provided callback

callback [callable] callable which will be invoked when events of type evname are received on this listener's rx connection

args, kwargs [initial arguments which will be partially applied to] callback right now

`add_handler`(*evname, handler*)

Register an event handler for events of type *evname*. If a handler for *evname* already exists or if *evname* is in the unsubscribe list an error will be raised.

`bg_jobs`

Background jobs collection

`block_jobs`()

Block the event loop from processing background job events (useful for registering for job events - see *self.register_job*)

This will block the event loop thread permanently starting on the next received background job event. Be sure to run 'unblock_jobs' immediately after registering your job.

`connect`()

Connect and initialize all contained esl sockets (namely *self._rx_con* and *self._tx_con*)

`connected`()

Return a bool representing the aggregate cons status

`count_calls`()

Count the number of active calls hosted by the slave process

`count_failed`()

Return the failed session count

`default_handlers`

The map of default event handlers described by this listener

`disconnect`()

Shutdown this listener's bg thread and disconnect all esl sockets.

This method should not be called by the event loop thread or you may see an indefinite block!

`epoch`

Time first event was received from server

`get_id`(*e, default=None*)

Acquire the client/consumer (app) id for event **:var:'e'**

`get_new_con`(*server=None, port=None, auth=None, register_events=False, **kwargs*)

Return a new esl connection to the specified FS server and optionally subscribe to any events actively handled by this listener

server [string] fs server ip

port [string] port to connect on

auth [string] authorization username

register_events [bool] indicates whether or not the connection should be subscribed to receive all default events declared by the listener's 'default_handlers' map

kwargs : same as for .connection.Connection

con : Connection

ident (*host='unknown-host'*)
Pretty str repr of connection-like instances.

is_alive ()
Return bool indicating if listener is running (i.e. the background event loop is executing).

iter_cons ()
Return an iterator over all attributes of this instance which are esl connections.

lookup_sess (*e*)
The most basic handler template which looks up the locally tracked session corresponding to event *e* and updates it with event data

register_job (*event*, ***kwargs*)
Register for a job to be handled when the appropriate event arrives. Once an event corresponding to the job is received, the bgjob event handler will 'consume' it and invoke its callback.

event [ESL.ESLevent] as returned from an ESLConnection.bgapi call

kwargs [dict] same signatures as for Job.__init__

bj : an instance of Job (a background job)

remove_callback (*evname*, *ident*, *callback*)
Remove the callback object registered under **:var:'evname'** and **:var:'ident'**.

reset ()
Clear all internal stats and counters

start ()
Start this listener's event loop in a thread to start tracking the slave-server's state

status ()
Return the status of ESL connections in a dict A value of True indicates that the connection is active. Returns map of con names -> connected() bools.

unblock_jobs ()
Unblock the event loop from processing background job events

unsubscribe (*events*)
Unsubscribe this listener from an events of a cetain type

events [string or iterable] name of mod_event event type(s) you wish to unsubscribe from (FS server will not be told to send you events of this type)

uptime
Uptime in minutes as per last received event time stamp

wait (*timeout=None*)
Wait until the event loop thread terminates or timeout.

waitfor(*sess*, *varname*, *timeout=None*)
Wait on a boolen variable *varname* to be set to true for session *sess* as read from *sess.vars['varname']*. This call blocks until the attr is set to *True* most usually by a callback.

Do not call this from the event loop thread!

switchy.observe.**active_client**(**args*, ***kwds*)
A context manager which delivers an active *Client* containing a started *EventListener* with applications loaded that were passed in the *apps* map

switchy.observe.**con_repr**(*self*)
Repr with a [<connection-status>] slapped in

switchy.observe.**get_listener**(*host*, *port='8021'*, *auth='ClueCon'*, *mng=None*, *mng_init=None*, ***kwargs*)
Listener factory which can be used to load a local instance or a shared proxy using *multiprocessing.managers*

switchy.observe.**get_pool**(*contacts*, ***kwargs*)
Construct and return a slave pool from a sequence of contact information.

4.4 Models

Models representing FreeSWITCH entities

class switchy.models.**Call**(*uuid*, *session*)
A collection of sessions which a compose a call

append(*sess*)
Append a session to this call and update the ref to the last recently added session

first
A reference to the session making up the initial leg of this call

get_peer(*sess*)
Convenience helper which can determine whether *sess* is one of *first* or *last* and returns the other when the former is true

hangup()
Hangup up this call

last
A reference to the session making up the final leg of this call

class switchy.models.**Events**(*event=None*)
Event collection which for most intents and purposes should quack like a collections.deque. Data lookups are delegated to the internal deque of events in lilo order.

get(*key*, *default=None*)
Return default if not found Should be faster then handling the key error?

pprint(*index=0*)
Print serialized event data in chronological order to stdout

update(*event*)
Append an ESL.ESLEvent

class switchy.models.**Job**(*event*, *sess_uuid=None*, *callback=None*, *client_id=None*, *kwargs={}*)
A background job future. The interface closely matches *multiprocessing.pool.AsyncResult*.

Parameters

- **uuid** (*str*) – job uuid returned directly by SOCKET_DATA event
- **sess_uuid** (*str*) – optional session uuid if job is associated with an active FS session

fail(*resp*, **args*, ***kwargs*)
Fail this job optionally adding an exception for its result

get(*timeout=None*)
Get the result for this job waiting up to *timeout* seconds. Raises *TimeoutError* on if job does complete within alotted time.

ready()
Return bool indicating whether job has completed

result
The final result

successful()
Return bool determining whether job completed without error

update(*event*)
Update job state/data using an event

wait(*timeout=None*)
Wait until job has completed or *timeout* has expired

class switchy.models.**Session**(*event*, *uuid=None*, *con=None*)
Session API and state tracking.

breakmedia()
Stop playback of media on this session and move on in the dialplan.

bridge(*dest_url=None*, *profile=None*, *gateway=None*, *proxy=None*, *params=None*)
Bridge this session using *uuid_broadcast* (so async). By default the current profile is used to bridge to the SIP Request-URI.

broadcast(*path*, *leg='‘*, *hangup_cause=None*)
Execute an application on a chosen leg(s) with optional hangup afterwards. `uuid_broadcast <uuid> app[![hangup_cause]]::args [aleg|bleg|both]`

bypass_media(*state*)
Re-invite a bridged node out of the media path for this session

clear_tasks()
Clear all scheduled tasks for this session.

deflect(*uri*)
Send a refer to the client. The only parameter should be the SIP URI to contact (with or without “sip:”):

```
<action application="deflect" data="sip:someone@somewhere.com" />
```

echo()
Echo back all audio recieved

get(*key*, *default=None*)
Get latest event header field for *key*.

hangup(*cause='NORMAL_CLEARING'*)
Hangup this session with the provided *cause* hangup type keyword.

host
Return the hostname/ip address for the host which this session is currently active

is_inbound()
Return bool indicating whether this is an inbound session

is_outbound()
Return bool indicating whether this is an outbound session

log
Local logger instance.

mute(*direction='write'*, *level=1*)
Mute the current session. *level* determines the degree of comfort noise to generate if > 1.

park()
Park this session

playback(*args*, *start_sample=None*, *endless=False*, *leg='aleg'*, *params=None*)
Playback a file on this session

Parameters

- **args** (`str or tuple`) – arguments or path to audio file for playback app
- **leg** (`str`) – call leg to transmit the audio on

record(*action*, *path*, *rx_only=True*)
Record audio from this session to a local file on the slave filesystem using the uuid_record command:

```
uuid_record <uuid> [start|stop|mask|unmask] <path> [<limit>]
```

respond(*response*)
Respond immediately with the following *response* code. see the FreeSWITCH respond dialplan application

sched_dtmf(*delay*, *sequence*, *tone_duration=None*)
Schedule dtmf sequence to be played on this channel.

Parameters

- **delay** (`float`) – scheduled future time when dtmf tones should play
- **sequence** (`str`) – sequence of dtmf digits to play

sched_hangup(*timeout*, *cause='NORMAL_CLEARING'*)
Schedule this session to hangup after *timeout* seconds.

send_dtmf(*sequence*, *duration='w'*)
Send a dtmf sequence with constant tone durations

setvar(*var*, *value*)
Set variable to value

setvars(*params*)
Set all variables in map *params* with a single command

start_record(*path*, *rx_only=False*, *stereo=False*, *rate=16000*)
Record audio from this session to a local file on the slave filesystem using the record_session cmd. By default recordings are sampled at 16kHz.

stop_record(*path='all'*, *delay=0*)
Stop recording audio from this session to a local file on the slave filesystem using the stop_record_session cmd.

time
Time stamp for the most recent received event

unmute(***kwargs*)
Unmute the write buffer for this session

unsetvar(*var*)
Unset a channel var.

update(*event*)
Update state/data using an ESL.ESLEvent

uptime
Time elapsed since the *Session.create_ev* to the most recent received event.

4.5 Distributed cluster tools

Manage pools of freeswitch slaves

class switchy.distribute.**MultiEval**(*slaves*, *delegator=<type 'itertools.cycle'>*, *accessor='.'*)
Invoke arbitrary python expressions on a collection of objects

attrs(*obj*)
Cache of obj attributes since python has no built in for getting them all...

evals(*expr*, ***kwargs*)
Evaluate expression on all slave sub-components (Warning: this is the slowest call)

expr: str python expression to evaluate on slave components

folder(*func*, *expr*, ***kwargs*)
Same as reducer but takes in a binary function

partial(*expr*, ***kwargs*)
Return a partial which will eval bytcode compiled from *expr*

reducer(*func*, *expr*, *itertype=''*, ***kwargs*)
Reduces the iter retured by *evals(expr)* into a single value using the reducer *func*

switchy.distribute.**SlavePool**(*slaves*)
A slave pool for controlling multiple (*Client*, *EventListener*) pairs with ease

4.6 Synchronous Calling

Make calls synchronously

switchy.sync.**sync_caller**(**args*, ***kwds*)
Deliver a provisioned synchronous caller function.

A caller let's you make a call synchronously returning control once it has entered a stable state. The caller returns the active originating *Session* and a *waitfor* blocker method as output.

4.7 Built-in Apps

Built-in applications

class switchy.apps.**AppManager**(*pool*, *ppfuncargs=None*, ***kwargs*)
Manage apps over a cluster/slavepool.

iterapps()
: Iterable over all unique contained subapps

load_app(*app*, *app_id=None*, *ppkwargs=None*, *with_measurers=()*)
: Load and activate an app for use across all slaves in the cluster.

load_multi_app(*apps_iter*, *app_id=None*, ***kwargs*)
: Load a "composed" app (multiple apps using a single app name/id) by providing an iterable of (app, prepost_kwargs) tuples. Whenever the app is triggered from and event loop all callbacks from all apps will be invoked in the order then were loaded here.

`switchy.apps.`**app**(**args*, ***kwargs*)
: Decorator to register switchy application classes. Example usage:

```
@app
class CoolAppController(object):
    pass

# This will register the class as a switchy app.
# The name of the app defaults to `class.__name__`.
# The help for the app is taken from `class.__doc__`.

# You can also provide an alternative name via a
# decorator argument:

@app('CoolName')
class CoolAppController(object):
    pass

# or with a keyword arg:

@app(name='CoolName')
class CoolAppController(object):
    pass
```

`switchy.apps.`**get**(*name*)
: Get a registered app by name or None if one isn't registered.

`switchy.apps.`**groupbymod**()
: Return an iterable which delivers tuples (<modulename>, <apps_subiter>)

`switchy.apps.`**iterapps**()
: Iterable over all registered apps.

`switchy.apps.`**load**(*packages=()*, *imp_excs=('pandas'*, *)*)
: Load by importing all built-in apps along with any apps found in the provided *packages* list.

  Parameters packages (*str | module*) – package (names or actual modules)

  Return type dict[str, types.ModuleType]

`switchy.apps.`**register**(*cls*, *name=None*)
: Register an app in the global registry

4.7.1 Load testing

Call generator app for load testing

class `switchy.apps.call_gen.`**Originator**(*slavepool*, *debug=False*, *auto_duration=True*, *app_id=None*, ***kwargs*)
: An auto-dialer built for stress testing.

check_state(*ident*)
Compare current state to ident

hard_hupall()
Hangup all calls for all slaves, period, even if they weren't originated by this instance and stop the burst loop.

hupall()
Send the 'hupall' command to hangup all active calls.

is_alive()
Indicate whether the call burst thread is up

load_app(*app*, *app_id=None*, *ppkwargs={}*, *weight=1*, *with_metrics=True*)
Load a call control app for use across the entire slave cluster.

If *app* is an instance then it's state will be shared by all slaves. If it is a class then new instances will be instantiated for each *Client-Observer* pair and thus state will have per slave scope.

max_rate
The maximum *rate* value which can be set. Setting *rate* any higher will simply clip to this value.

originate_cmd
Originate str used for making calls

setup()
Apply load test settings on the slave server

shutdown()
Shutdown this originator instance and hanging up all active calls and triggering the burst loop to exit.

start()
Start the originate burst loop by starting and/or notifying a worker thread to begin. Changes state INITIAL | STOPPED -> ORIGINATING.

state
The current operating state as a string

stop()
Stop originate loop if currently originating sessions. Change state ORIGINATING -> STOPPED

stopped()
Return bool indicating if in the stopped state.

waitwhile(*state_or_predicate=<function <lambda>>*, ***kwargs*)
If *state_or_predicate' is a func, block until it evaluates to 'False*. If it is a *str* block until the internal state matches that value. The default predicate waits for all calls to end and for activation of the "STOPPED" state. See *switchy.utils.waitwhile* for more details on predicate usage.

class switchy.apps.call_gen.**State**(*state=0*)
Enumeration to represent the originator state machine

class switchy.apps.call_gen.**WeightedIterator**(*counter=None*)
Pseudo weighted round robin iterator. Delivers items interleaved in weighted order.

cycle()
Endlessly iterates the most up to date keys in *counts*. Allows for real-time weight updating from another thread.

switchy.apps.call_gen.**get_originator**(*contacts*, **args*, ***kwargs*)
Originator factory

`switchy.apps.call_gen.`**`limiter`**(*pairs*)
Yield slave pairs up until a slave has reached a number of calls less then or equal to it's predefined capacity limit

4.7.2 Measurement Collection

CDR app for collecting signalling latency and performance stats.

class `switchy.apps.measure.cdr.`**`CDR`**
Collect call detail record info including call oriented event time stamps and and active sessions data which can be used for per call metrics computations.

`log_stats`(*sess*, *job*)
Append measurement data only once per call

`on_create`(*sess*)
Store total (cluster) session count at channel create time

`switchy.apps.measure.cdr.`**`call_metrics`**(*df*)
Default call measurements computed from data retrieved by the *CDR* app.

System stats collection using **'psutil'_**

class `switchy.apps.measure.sys.`**`SysStats`**(*psutil*, *rpyc=None*)
A switchy app for capturing system performance stats during load test using the **'psutil'_** module.

An instance of this app should be loaded if rate limited data gathering is to be shared across multiple slaves (threads).

`switchy.apps.measure.sys.`**`sys_stats`**(*df*)
Reindex on the call index to allign with call metrics data and interpolate.

4.7.3 Media testing

Common testing call flows

class `switchy.apps.players.`**`PlayRec`**
Play a recording to the callee and record it onto the local file system

This app can be used in tandem with MOS scoring to verify audio quality. The filename provided must exist in the FreeSWITCH sounds directory such that ${FS_CONFIG_ROOT}/${sound_prefix}/<category>/<filename> points to a valid wave file.

`on_stop`(*sess*)
On stop either trigger a new playing of the signal if more iterations are required or hangup the call. If the current call is being recorded schedule the recordings to stop and expect downstream callbacks to schedule call teardown.

`trigger_playback`(*sess*)
Trigger clip playback on the given session by doing the following: - Start playing a silence stream on the peer session - This will in turn trigger a speech playback on this session in the "PLAYBACK_START" callback

class `switchy.apps.players.`**`RecInfo`**(*host*, *caller*, *callee*)

`callee`
Alias for field number 2

`caller`
Alias for field number 1

host
Alias for field number 0

class `switchy.apps.players.`**`TonePlay`**
Play a 'milli-watt' tone on the outbound leg and echo it back on the inbound

Dtmf tools

class `switchy.apps.dtmf.`**`DtmfChecker`**
Play dtmf tones as defined by the iterable attr *sequence* with tone *duration*. Verify the rx sequence matches what was transmitted.

For each session which is answered start a sequence check. For any session that fails digit matching store it locally in the *failed* attribute.

Bert testing

class `switchy.apps.bert.`**`Bert`**
Call application which runs the bert test application on both legs of a call

See the docs for mod_bert and discussion by the author here.

hangup_on_error
Toggle whether to hangup calls when a bert test fails

on_lost_sync(*sess*)
Increment counters on synchronization failure

The following stats can be retrieved using the latest version of mod_bert:

> sync_lost_percent - Error percentage within the analysis window sync_lost_count - How many times sync has been lost cng_count - Counter of comfort noise packets err_samples - Number of samples that did not match the sequence

on_park(*sess*)
Knows how to get us riled up

on_timeout(*sess*)
Mark session as bert time out

two_sided
Toggle whether to run the *bert_test* application on all sessions of the call. Leaving this *False* means all other legs will simply run the *echo* application.

4.8 Command Builders

Command wrappers and helpers

`switchy.commands.`**`build_originate_cmd`**(*dest_url*, *uuid_str=None*, *profile='external'*, *gateway=None*, *app_name='park'*, *app_arg_str=''*, *dp_exten=None*, *dp_type='xml'*, *dp_context='default'*, *proxy=None*, *endpoint='sofia'*, *timeout=60*, *caller_id='Mr_Switchy'*, *caller_id_num='1112223333'*, *codec='PCMU'*, *abs_codec=''*, *xheaders=None*, ***kwargs*)
Return a formatted *originate* command string conforming to the syntax dictated by mod_commands of the form:

originate <call url> <exten>|&<application_name>(<app_args>) [<dialplan>] [<context>] [<cid_name>] [<cid_num>] [<timeout_sec>]

dest_url [str] call destination url with format <username_uri>@<domain>:<port>

profile [str] sofia profile (UA) name to use for making outbound call

dp_extension: str destination dp extension where the originating session (a-leg) will processed just after the call is answered

etc...

originate command [string or callable] full cmd string if uuid_str is not None, else callable f(uuid_str) -> full cmd string

4.9 Utils

handy utilities

exception `switchy.utils.`**`APIError`**
ESL api error

exception `switchy.utils.`**`ConfigurationError`**
Config error

`switchy.utils.`**`DictProxy`**(*d*, *extra_attrs={}*)
A dictionary proxy object which provides attribute access to elements

exception `switchy.utils.`**`ESLError`**
An error pertaining to the connection

exception `switchy.utils.`**`TimeoutError`**
Timing error

class `switchy.utils.`**`Timer`**(*timer=None*)
Simple timer that reports an elapsed duration since the last reset.

`elapsed`()
Returns the elapsed time since the last reset

`last_time`
Last time the timer was reset

`reset`()
Reset the timer start point to now

`switchy.utils.`**`compose`**(*func_1*, *func_2*)
(f1, f2) -> function The function returned is a composition of f1 and f2.

`switchy.utils.`**`dirinfo`**(*inst*)
Return common info useful for dir output

`switchy.utils.`**`event2dict`**(*event*)
Return event serialized data in a python dict Warning: this function is kinda slow!

`switchy.utils.`**`get_args`**(*func*)
Return the argument names found in func's signature in a tuple

Returns the argnames, kwargnames defined by func

Return type tuple

`switchy.utils.`**`get_event_time`**(*event*, *epoch=0.0*)
Return micro-second time stamp value in seconds

`switchy.utils.`**`get_logger`**(*name=None*)
Return the package log or a sub-log for *name* if provided.

`switchy.utils.`**`get_name`**`(`*obj*`)`
Return a name for object checking the usual places

`switchy.utils.`**`is_callback`**`(`*func*`)`
Check whether func is valid as a callback

`switchy.utils.`**`iter_import_submods`**`(`*packages*, *recursive=False*, *imp_excs=()*`)`
Iteratively import all submodules of a module, including subpackages with optional recursion.

Parameters **`package`** (*`str | module`*) – package (name or actual module)

Return type (dict[str, types.ModuleType], dict[str, ImportError])

`switchy.utils.`**`log_to_stderr`**`(`*level=None*`)`
Turn on logging and add a handler which writes to stderr

`switchy.utils.`**`ncompose`**`(`**funcs*`)`
Perform n-function composition

`switchy.utils.`**`param2header`**`(`*name*`)`
Return the appropriate event header name corresponding to the named parameter *name* which should be used when the param is received as a header in event data.

Most often this is just the original parameter name with a `'variable_'` prefix. This is pretty much a shitty hack (thanks goes to FS for the asymmetry in variable referencing...)

`switchy.utils.`**`pstr`**`(`*self*, *host='unknown-host'*`)`
Pretty str repr of connection-like instances.

`switchy.utils.`**`uncons`**`(`*first*, **rest*`)`
Unpack args into first element and tail as tuple

`switchy.utils.`**`uuid`**`()`
Return a new uuid1 string

`switchy.utils.`**`waitwhile`**`(`*predicate*, *timeout=inf*, *period=0.1*, *exc=True*`)`
Block until *predicate* evaluates to *False*.

Parameters

- **`predicate`** (*`function`*) – predicate function
- **`timeout`** (*`float`*) – time to wait in seconds for predicate to eval False
- **`period`** (*`float`*) – poll loop sleep period in seconds

Raises **`TimeoutError`** – if predicate does not eval to False within *timeout*

`switchy.utils.`**`xheaderify`**`(`*header_name*`)`
Prefix the given name with the freeswitch xheader token thus transforming it into an fs xheader variable

4.10 API Reference

Note: This reference is not entirely comprehensive and is expected to change.

4.10.1 Connection wrapper

A thread safe (plus more) wrapper around the ESL swig module's *ESLConnection* type is found in connection.py.

4.10.2 Observer components

The core event processing loop and logic and *`Client`* interface can be found in observe.py. There are also some synchronous helpers hidden within.

4.10.3 Call Control Apps

All the built in apps can be found in the *`switchy.apps`* subpackage.

4.10.4 Model types

The Models api holds automated wrappers for interacting with different *FreeSWITCH* channel and session objects as if they were local instances.

- *`Session`* - represents a *FreeSWITCH session* entity and provides a rich method api for control using call management commands.
- *`Job`* - provides a synchronous interface for background job handling.

4.10.5 Cluster tooling

Extra helpers for managing a *FreeSWITCH* process cluster.

- *`MultiEval`* - Invoke arbitrary python expressions on a collection of objects.
- *`SlavePool`* - a subclass which adds oberver component helper methods.

4.11 Quick-Start - Originating a single call

Assuming you've gone through the required deployment steps to setup at least one slave, initiating a call becomes very simple using the Switchy command line:

```
$ switchy run vm-host sip-cannon --profile external --proxy myproxy.com --rate 1 --limit 1 --max-offe

...

Aug 26 21:59:01 [INFO] switchy cli.py:114 : Slave sip-cannon.qa.sangoma.local SIP address is at 10.10
Aug 26 21:59:01 [INFO] switchy cli.py:114 : Slave vm-host.qa.sangoma.local SIP address is at 10.10.8
Aug 26 21:59:01 [INFO] switchy cli.py:120 : Starting load test for server dut-008.qa.sangoma.local at
<Originator: active-calls=0 state=INITIAL total-originated-sessions=0 rate=1 limit=1 max-offered=1 du

...

<Originator: active-calls=1 state=STOPPED total-originated-sessions=1 rate=1 limit=1 max-offered=1 du
Waiting on 1 active calls to finish
Waiting on 1 active calls to finish
Waiting on 1 active calls to finish
Waiting on 1 active calls to finish
Load test finished!
```

The Switchy *run* sub-command takes several options and a list of slaves (or at least one) IP address or hostname. In this example switchy connected to the specified slaves, found the specified SIP profile and initiated a single call with a duration of 5 seconds to the device under test (set with the *proxy* option).

For more information on the switchy command line see here.

4.11.1 Originating a single call programatically from Python

Making a call with switchy is quite simple using the built-in `sync_caller()` context manager. Again, if you've gone through the required deployment steps, initiating a call becomes as simple as a few lines of python code

```
1  from switchy import sync_caller
2  from switchy.apps.players import TonePlay
3
4  # here '192.168.0.10' would be the address of the server running a
5  # FS process to be used as the call generator
6  with sync_caller('192.168.0.10', apps={"tone": TonePlay}) as caller:
7
8      # initiates a call to the originating profile on port 5080 using
9      # the `TonePlay` app and block until answered / the originate job completes
10     sess, waitfor = caller('Fred@{}:{}'.format(caller.client.host, 5080), "tone")
11     # let the tone play a bit
12     time.sleep(5)
13     # tear down the call
14     sess.hangup()
```

The most important lines are the *with* statement and line 10. What happens behind the scenes here is the following:

- at the *with*, necessary internal Switchy components are instantiated in memory and connected to a *FreeSWITCH* process listening on the *fsip* ESL ip address.
- at the *caller()*, an `originate()` command is invoked asynchronously via a `bgapi()` call.
- the background `Job` returned by that command is handled to completion **synchronously** wherein the call blocks until the originating session has reached the connected state.
- the corresponding origininating `Session` is returned along with a reference to a `switchy.observe.EventListener.waitfor()` blocker method.
- the call is kept up for 1 second and then `hungup`.
- internal Switchy components are disconnected from the *FreeSWITCH* process at the close of the *with* block.

Note that the *sync_caller* api is not normally used for stress testing as it used to initiate calls *synchronously*. It becomes far more useful when using *FreeSWITCH* for functional testing using your own custom call flow apps.

4.11.2 Example source code

Some more extensive examples are found in the unit tests sources :

Listing 4.1: test_sync_call.py

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
"""
Tests for synchronous call helper
"""
import time
from switchy import sync_caller
from switchy.apps.players import TonePlay, PlayRec

```

```python

def test_toneplay(fsip):
    '''Test the synchronous caller with a simple toneplay
    '''
    with sync_caller(fsip, apps={"TonePlay": TonePlay}) as caller:
        # have the external prof call itself by default
        assert 'TonePlay' in caller.app_names
        sess, waitfor = caller(
            "doggy@{}:{}".format(caller.client.host, 5080),
            'TonePlay',
            timeout=3,
        )
        assert sess.is_outbound()
        time.sleep(1)
        sess.hangup()
        time.sleep(0.1)
        assert caller.client.listener.count_calls() == 0


def test_playrec(fsip):
    '''Test the synchronous caller with a simulated conversation using the the
    `PlayRec` app. Currently this test does no audio checking but merely
    verifies the callback chain is invoked as expected.
    '''
    with sync_caller(fsip, apps={"PlayRec": PlayRec}) as caller:
        # have the external prof call itself by default
        caller.apps.PlayRec['PlayRec'].rec_rate = 1
        sess, waitfor = caller(
            "doggy@{}:{}".format(caller.client.host, 5080),
            'PlayRec',
            timeout=10,
        )
        waitfor(sess, 'recorded', timeout=15)
        waitfor(sess.call.get_peer(sess), 'recorded', timeout=15)
        assert sess.call.vars['record']
        time.sleep(1)
        assert sess.hungup


def test_alt_call_tracking_header(fsip):
    '''Test that an alternate `EventListener.call_tracking_header` (in this
    case using the 'Caller-Destination-Number' channel variable) can be used
    to associate sessions into calls.
    '''
    with sync_caller(fsip) as caller:
        # use the destination number as the call association var
        caller.client.listener.call_tracking_header = 'Caller-Destination-Number'
        dest = 'doggy'
        # have the external prof call itself by default
        sess, waitfor = caller(
            "{}@{}:{}".format(dest, caller.client.host, 5080),
            'TonePlay',  # the default app
            timeout=3,
        )
        assert sess.is_outbound()
        # call should be indexed by the req uri username
        assert dest in caller.client.listener.calls
        call = caller.client.listener.calls[dest]
        time.sleep(1)
        assert call.first is sess
        assert call.last
        call.hangup()
        time.sleep(0.1)
        assert caller.client.listener.count_calls() == 0


def test_untracked_call(fsip):
    with sync_caller(fsip) as caller:
        # use an invalid chan var for call tracking
        caller.client.listener.call_tracking_header = 'doggypants'
        # have the external prof call itself by default
        sess, waitfor = caller(
            "{}@{}:{}".format('jonesy', caller.client.host, 5080),
            'TonePlay',  # the default app
            timeout=3,
        )
        # calls should be created for both inbound and outbound sessions
        # since our tracking variable is nonsense
        l = caller.client.listener
        # assert len(l.sessions) == len(l.calls) == 2
        assert l.count_sessions() == l.count_calls() == 2
        sess.hangup()
        time.sleep(0.1)
        # no calls or sessions should be active
        assert l.count_sessions() == l.count_calls() == 0
        assert not l.sessions and not l.calls
```

Run manually

You can run this code from the unit test directory quite simply:

```
>>> from tests.test_sync_call import test_toneplay
>>> test_toneplay('fs_slave_hostname')
```

Run with pytest

If you have `pytest` installed you can run this test like so:

```
$ py.test --fshost='fs_slave_hostname' tests/test_sync_caller
```

Implementation details

The implementation of *sync_caller()* is shown below and can be referenced alongside the Internals tutorial to gain a better understanding of the inner workings of Switchy's api:

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
"""
Make calls synchronously
"""
from contextlib import contextmanager
from switchy.apps.players import TonePlay
```

```python
from switchy.observe import active_client


@contextmanager
def sync_caller(host, port='8021', password='ClueCon',
                apps={'TonePlay': TonePlay}):
    '''Deliver a provisioned synchronous caller function.

    A caller let's you make a call synchronously returning control once
    it has entered a stable state. The caller returns the active originating
    `Session` and a `waitfor` blocker method as output.
    '''
    with active_client(host, port=port, auth=password, apps=apps) as client:

        def caller(dest_url, app_name, timeout=30, waitfor=None,
                   **orig_kwargs):
            # override the channel variable used to look up the intended
            # switchy app to be run for this call
            if caller.app_lookup_vars:
                client.listener.app_id_vars.extend(caller.app_lookup_vars)

            job = client.originate(dest_url, app_id=app_name, **orig_kwargs)
            job.get(timeout)
            if not job.successful():
                raise job.result
            call = client.listener.sessions[job.sess_uuid].call
            orig_sess = call.first  # first sess is the originator
            if waitfor:
                var, time = waitfor
                client.listener.waitfor(orig_sess, var, time)

            return orig_sess, client.listener.waitfor

        # attach apps handle for easy interactive use
        caller.app_lookup_vars = []
        caller.apps = client.apps
        caller.client = client
        caller.app_names = client._apps.keys()
        yield caller
```

4.12 Call Applications

switchy supports writing and composing call control *applications* written in pure Python. An *app* is simply a namespace which defines **a set of event callbacks** [1].

Apps are somewhat analogous to extensions in *FreeSWITCH*'s XML dialplan interface and can similarly be activated using any event header *or* channel variable value of your choosing. Callbacks are invoked based on the recieved event type.

Apps can be implemented each as a standalone Python namespace which can hold state and be mutated at runtime. This allows for all sorts of dynamic call processing logic. *Apps* can also be shared across a *FreeSWITCH* process cluster allowing for centralized call processing overtop a scalable service system.

Applications are *loaded* either using a `Client` or, in the case of an *switchy* cluster Service, an `AppManager` instance.

[1] Although this may change in the future with the introduction of native asyncio coroutines in Python 3.5.

4.12.1 API

Apps are usually implemented as plain old Python classes which contain methods decorated using the `switchy.marks` module.

Currently the marks supported would be one of:

```
@event_callback("EVENT_NAME")
@handler("EVENT_NAME")
```

Where *EVENT_NAME* is any of the strings supported by the ESL event type list.

Additionally, app types can support a `prepost()` callable which serves as a setup/teardown fixture mechanism for the app to do pre/post app loading execution. It can be either of a function or generator.

Note: For examples using `prepost()` see the extensive set of built-in apps under *`switchy.apps`*.

Event Callbacks

`event_callbacks` are methods which typically receive a type from *`switchy.models`* as their first (and only) argument. This type is most often a *`Session`*.

Note: Technically the method will receive whatever is returned as the 2nd value from the preceeding event *handler* looked up in the event processing loop, but this is an implementation detail and may change in the future.

Here is a simple callback which counts the number of answered sessions in a global:

```
import switchy

num_calls = 0

@switchy.event_callback('CHANNEL_ANSWER')
def counter(session):
    global num_calls
    num_calls += 1
```

Note: This is meant to be a simple example and not actually implemented for practical use. *`switchy.observe.EventListener.count_calls()`* exists for this very purpose.

Event Handlers

An event handler is any callable marked by `handler()` which is expected to handle a received *ESLEvent* object and process it within the *`EventListener`* event loop. It's function signature should expect a single argument, that being the received event.

Example handlers can be found in the *`EventListener`* such as the default *CHANNEL_ANSWER* handler

```
    def _handle_answer(self, e):
        '''Handle answer events

        Returns
        --------
        sess : session instance corresponding to uuid
        '''
        uuid = e.getHeader('Unique-ID')
        sess = self.sessions.get(uuid, None)
        if sess:
            self.log.debug('answered session {} with call direction {}'
                           .format(uuid,  e.getHeader('Call-Direction')))
            sess.answered = True
            self.total_answered_sessions += 1
            sess.update(e)
            return True, sess
        else:
            self.log.warn('Skipping answer of {}'.format(uuid))
            return False, None
```

As you can see a knowledge of the underlying ESL SWIG python package usually is required for *handler* implementations.

4.12.2 Examples

TonePlay

As a first example here is the `TonePlay` app which is provided as a built-in for Switchy

```
class TonePlay(object):
    """Play a 'milli-watt' tone on the outbound leg and echo it back
    on the inbound
    """
    @event_callback('CHANNEL_PARK')
    def on_park(self, sess):
        if sess.is_inbound():
            sess.answer()

    @event_callback("CHANNEL_ANSWER")
    def on_answer(self, sess):
        # inbound leg simply echos back the tone
        if sess.is_inbound():
            sess.echo()

        # play infinite tones on calling leg
        if sess.is_outbound():
            sess.broadcast('playback::{loops=-1}tone_stream://%(251,0,1004)')
```

`Clients` who load this app will originate calls wherein a simple tone is played infinitely and echoed back to the caller until each call is hung up.

Proxier

An example of the *proxy dialplan* can be implemented quite trivially:

```
import switchy

class Proxier(object):
    @switchy.event_callback('CHANNEL_PARK')
    def on_park(self, sess):
        if sess.is_inbound():
            sess.bridge(dest_url="${sip_req_user}@${sip_req_host}:${sip_req_port}")
```

CDR

The measurement application used by the `Originator` to gather stress testing performance metrics from call detail records:

```
class CDR(object):
    """Collect call detail record info including call oriented event time
    stamps and and active sessions data which can be used for per call metrics
    computations.
    """
    fields = [
        ('switchy_app', 'S50'),
        ('hangup_cause', 'S50'),
        ('caller_create', 'float64'),
        ('caller_answer',  'float64'),
        ('caller_req_originate', 'float64'),
        ('caller_originate', 'float64'),
        ('caller_hangup', 'float64'),
        ('job_launch', 'float64'),
        ('callee_create', 'float64'),
        ('callee_answer', 'float64'),
        ('callee_hangup', 'float64'),
        ('failed_calls', 'uint32'),
        ('active_sessions', 'uint32'),
        ('erlangs', 'uint32'),
    ]

    operators = {
        'call_metrics': call_metrics,
        # 'call_types': call_types,
        # 'hcm': hcm,
    }

    def __init__(self):
        self.log = utils.get_logger(__name__)
        self._call_counter = itertools.count(0)

    def new_storer(self):
        return DataStorer(self.__class__.__name__, dtype=self.fields)

    def prepost(self, listener, storer=None, pool=None, orig=None):
        self.listener = listener
        self.orig = orig
        # create our own storer if we're not loaded as a `Measurer`
        self._ds = storer if storer else self.new_storer()
        self.pool = weakref.proxy(pool) if pool else self.listener

    @property
    def storer(self):
        return self._ds

    @event_callback('CHANNEL_CREATE')
    def on_create(self, sess):
        """Store total (cluster) session count at channel create time
        """
        call_vars = sess.call.vars
        # call number tracking
        if not call_vars.get('call_index', None):
            call_vars['call_index'] = next(self._call_counter)
        # capture the current erlangs / call count
        call_vars['session_count'] = self.pool.count_sessions()
        call_vars['erlangs'] = self.pool.count_calls()

    @event_callback('CHANNEL_ORIGINATE')
    def on_originate(self, sess):
        # store local time stamp for originate
        sess.times['originate'] = sess.time
        sess.times['req_originate'] = time.time()

    @event_callback('CHANNEL_ANSWER')
    def on_answer(self, sess):
        sess.times['answer'] = sess.time

    @event_callback('CHANNEL_HANGUP')
    def log_stats(self, sess, job):
        """Append measurement data only once per call
        """
        sess.times['hangup'] = sess.time
        call = sess.call

        if call.sessions:  # still session(s) remaining to be hungup
            call.caller = call.first
            call.callee = call.last
            if job:
                call.job = job
            return  # stop now since more sessions are expected to hangup

        # all other sessions have been hungup so store all measurements
        caller = getattr(call, 'caller', None)
        if not caller:
            # most likely only one leg was established and the call failed
            # (i.e. call.caller was never assigned above)
            caller = sess

        callertimes = caller.times
        callee = getattr(call, 'callee', None)
        calleetimes = callee.times if callee else None

        pool = self.pool
        job = getattr(call, 'job', None)
        # NOTE: the entries here correspond to the listed `CDR.fields`
        rollover = self._ds.append_row((
            caller.appname,
            caller['Hangup-Cause'],
            callertimes['create'],  # invite time index
            callertimes['answer'],
            callertimes['req_originate'],  # local time stamp
            callertimes['originate'],
            callertimes['hangup'],
            # 2nd leg may not be successfully established
            job.launch_time if job else None,
            calleetimes['create'] if callee else None,
            calleetimes['answer'] if callee else None,
            calleetimes['hangup'] if callee else None,
            pool.count_failed(),
            call.vars['session_count'],
            call.vars['erlangs'],
        ))
        if rollover:
            self.log.debug('wrote data to disk')
```

It simply inserts the call record data on hangup once for each *call*.

PlayRec

This more involved application demonstrates *FreeSWITCH*'s ability to play and record rtp streams locally which can be used in tandem with MOS to do audio quality checking:

```
class PlayRec(object):
    '''Play a recording to the callee and record it onto the local file system

    This app can be used in tandem with MOS scoring to verify audio quality.
    The filename provided must exist in the FreeSWITCH sounds directory such
    that ${FS_CONFIG_ROOT}/${sound_prefix}/<category>/<filename> points to a
    valid wave file.
    '''
    timer = utils.Timer()

    def prepost(
        self,
        client,
        filename='ivr-founder_of_freesource.wav',
        category='ivr',
        clip_length=4.25,  # measured empirically for the clip above
        sample_rate=8000,
        iterations=1,  # number of times the speech clip will be played
        callback=None,
        rec_period=5.0,  # in seconds (i.e. 1 recording per period)
        rec_stereo=False,
    ):
        self.filename = filename
        self.category = category
        self.framerate = sample_rate
        self.clip_length = clip_length
        if callback:
            assert inspect.isfunction(callback), 'callback must be a function'
            assert len(inspect.getargspec(callback)[0]) == 1
        self.callback = callback
        self.rec_period = rec_period
        self.stereo = rec_stereo
        self.log = utils.get_logger(self.__class__.__name__)
        self.silence = 'silence_stream://0'  # infinite silence stream
        self.iterations = iterations
        self.tail = 1.0

        # slave specific
        soundsdir = client.cmd('global_getvar sounds_dir')
        self.soundsprefix = client.cmd('global_getvar sound_prefix')
        # older FS versions don't return the deep path
        if soundsdir == self.soundsprefix:
            self.soundsprefix = '/'.join((self.soundsprefix, 'en/us/callie'))

        self.recsdir = client.cmd('global_getvar recordings_dir')
        self.audiofile = '{}/{}/{}/{}'.format(
            self.soundsprefix, self.category, self.framerate, self.filename)
        self.call2recs = OrderedDict()
        self.host = client.host

        # self.stats = OrderedDict()

    def __setduration__(self, value):
        """Called when an originator changes it's `duration` attribute
        """
        if value == float('inf'):
            self.iterations, self.tail = value, 1.0
        else:
            self.iterations, self.tail = divmod(value, self.clip_length)
        if self.tail < 1.0:
            self.tail = 1.0

    @event_callback("CHANNEL_PARK")
    def on_park(self, sess):
        if sess.is_inbound():
            sess.answer()

    @event_callback("CHANNEL_ANSWER")
    def on_answer(self, sess):
        call = sess.call
        if sess.is_inbound():
            # rec the callee stream
            elapsed = self.timer.elapsed()
            if elapsed >= self.rec_period:
                filename = '{}/callee_{}.wav'.format(self.recsdir, sess.uuid)
                sess.start_record(filename, stereo=self.stereo)
                self.call2recs.setdefault(call.uuid, {})['callee'] = filename
                call.vars['record'] = True
                # mark all rec calls to NOT be hung up automatically
                # (see the `Originator`'s bj callback)
                call.vars['noautohangup'] = True
                self.timer.reset()

            # set call length
            call.vars['iterations'] = self.iterations
            call.vars['tail'] = self.tail

        if sess.is_outbound():
            if call.vars.get('record'):  # call is already recording
                # rec the caller stream
                filename = '{}/caller_{}.wav'.format(self.recsdir, sess.uuid)
                sess.start_record(filename, stereo=self.stereo)
                self.call2recs.setdefault(call.uuid, {})['caller'] = filename
            else:
                self.trigger_playback(sess)

        # always enable a jitter buffer
        # sess.broadcast('jitterbuffer::60')
    @event_callback("PLAYBACK_START")
    def on_play(self, sess):
        fp = sess['Playback-File-Path']
        self.log.debug("Playing file '{}' for session '{}'"
                       .format(fp, sess.uuid))

        self.log.debug("fp is '{}'".format(fp))
        if fp == self.audiofile:
            sess.vars['clip'] = 'signal'
        elif fp == self.silence:
            # if playing silence tell the peer to start playing a signal
            sess.vars['clip'] = 'silence'
            peer = sess.call.get_peer(sess)
            if peer:  # may have already been hungup
                peer.breakmedia()
                peer.playback(self.audiofile)

    @event_callback("PLAYBACK_STOP")
    def on_stop(self, sess):
        '''On stop either trigger a new playing of the signal if more
        iterations are required or hangup the call.
        If the current call is being recorded schedule the recordings to stop
        and expect downstream callbacks to schedule call teardown.
        '''
        self.log.debug("Finished playing '{}' for session '{}'".format(
                      sess['Playback-File-Path'], sess.uuid))
        if sess.vars['clip'] == 'signal':
            vars = sess.call.vars
            vars['playback_count'] += 1

            if vars['playback_count'] < vars['iterations']:
                sess.playback(self.silence)
            else:
                # no more clips are expected to play
                if vars.get('record'):  # stop recording both ends
                    tail = vars['tail']
                    sess.stop_record(delay=tail)
                    peer = sess.call.get_peer(sess)
                    if peer:  # may have already been hungup
                        # infinite silence must be manually killed
                        peer.breakmedia()
                        peer.stop_record(delay=tail)
                else:
                    # hangup calls not being recorded immediately
                    self.log.debug("sending hangup for session '{}'"
                                   .format(sess.uuid))
                    if not sess.hungup:
                        sess.sched_hangup(0.5)  # delay hangup slightly

    def trigger_playback(self, sess):
        '''Trigger clip playback on the given session by doing the following:
        - Start playing a silence stream on the peer session
        - This will in turn trigger a speech playback on this session in the
        "PLAYBACK_START" callback
        '''
        peer = sess.call.get_peer(sess)
        peer.playback(self.silence)  # play infinite silence
        peer.vars['clip'] = 'silence'
        # start counting number of clips played
        sess.call.vars['playback_count'] = 0

    @event_callback("RECORD_START")
    def on_rec(self, sess):
        self.log.debug("Recording file '{}' for session '{}'".format(
            sess['Record-File-Path'], sess.uuid)
        )
        # mark this session as "currently recording"
        sess.vars['recorded'] = False
        # sess.setvar('timer_name', 'soft')

        # start signal playback on the caller
        if sess.is_outbound():
            self.trigger_playback(sess)

    @event_callback("RECORD_STOP")
    def on_recstop(self, sess):
        self.log.debug("Finished recording file '{}' for session '{}'".format(
            sess['Record-File-Path'], sess.uuid))
        # mark as recorded so user can block with `EventListener.waitfor`
        sess.vars['recorded'] = True
        if sess.hungup:
            self.log.warn(
                "sess '{}' was already hungup prior to recording completion?"
                .format(sess.uuid))

        # if sess.call.vars.get('record'):
        #     self.stats[sess.uuid] = sess.con.api(
        #         'json {{"command": "mediaStats", "data": {{"uuid": "{0}"}}}}'
        #         .format(sess.uuid)
        #     ).getBody()

        # if the far end has finished recording then hangup the call
        if sess.call.get_peer(sess).vars.get('recorded', True):
            self.log.debug("sending hangup for session '{}'".format(sess.uuid))
            if not sess.hungup:
                sess.sched_hangup(0.5)  # delay hangup slightly
            recs = self.call2recs[sess.call.uuid]

            # invoke callback for each recording
            if self.callback:
                self.callback(
                    RecInfo(self.host, recs['caller'], recs['callee'])
                )
```

For further examples check out the *apps* sub-package which also includes the very notorious *switchy.apps.call_gen.Originator*.

4.13 Building a cluster service

switchy supports building full fledged routing systems just like you can with *FreeSWITCH*'s XML dialplan but with the added benefit that you can use a centralized "dialplan" to control a *FreeSWITCH* process cluster.

This means call control logic can reside in one (or more) *switchy* process(es) running on a separate server allowing you to separate the *brains* and *logic* from the *muscle* and *functionality* when designing a scalable *FreeSWITCH* service system.

A service is very easy to create given a set of *deployed* *Freeswitch* processes:

```
from switchy import Service, event_callback

class Proxier(object):
    """Proxy all inbound calls to the destination specified in the SIP
    Request-URI.
    """
    @event_callback('CHANNEL_PARK')
    def on_park(self, sess):
        if sess.is_inbound():
            sess.bridge(dest_url="${sip_req_uri}")

s = Service(['FS_host1.com', 'FS_host2.com', 'FS_host3.com'])
s.apps.load_app(Proxier, app_id='default')
s.run()  # blocks forever
```

In this example all three of our *FreeSWITCH* servers load a *Proxier* app which simply bridges calls to the destination requested in the SIP Request-URI header. The *app_id='default'* kwarg is required to tell the internal event loop that this app should be used as the default (i.e. when no other app has consumed the event/session for processing).

4.13.1 *Flask*-like routing

Using the `Router` app we can define a routing system reminiscent of flask.

Let's start with an example of blocking certain codes:

```
from switchy.apps.routers import Router

router = Router(guards={
    'Call-Direction': 'inbound',
    'variable_sofia_profile': 'external'})

@router.route('00(.*)|011(.*)', response='407')
def reject_international(sess, match, router, response):
    sess.respond(response)
    sess.hangup()
```

There's a few things going on here:

- A `Router` is created with a *guard* `dict` which determines strict constraints on *event headers* which **must** be matched exactly for the `Router` to invoke registered (via `@route`) functions.
- We decorate a function, `reject_international`, which registers it to be invoked whenever an international number is dialed and will block such numbers with a SIP `407` response code.
- The first 3 arguments to `reject_international` are required, namely, `sess`, `match`, and `router` and correspond to the *`Session`*, re.MatchObject, and `Router` respectively.

In summmary, we can define *patterns* which must be matched against event headers before a particular *route function* will be invoked.

The signature for `Router.route` which comes from `PatternCaller` is:

`@route`(*pattern*, *field=None*, *kwargs*)

and works by taking in a *regex* `pattern`, an optional `field` (default is `'Caller-Destination-Number'`) and `kwargs`. The `pattern` must be matched against the `field` *event header* in order for the *route* to be called with `kwargs` (i.e. `reject_international(**kwargs)`).

Let's extend our example to include some routes which bridge differently based on the default `'Caller-Destination-Number'` *event header*:

```
from switchy.apps.routers import Router

router = Router({'Call-Direction': 'inbound'})

@router.route('00(.*)|011(.*)', response='407')
@router.route('1(.*)', gateway='long_distance_trunk')
@router.route('2[1-9]{3}$', out_profile='internal', proxy='salespbx.com')
@router.route('4[1-9]{3}$', out_profile='internal', proxy='supportpbx.com')
def bridge2dest(sess, match, router, out_profile=None, gateway=None,
                proxy=None, response=None):
    if response:
        sess.log.warn("Rejecting international call to {}".format(
            sess['Caller-Destination-Number']))
        sess.respond(response)
        sess.hangup()

    sess.bridge(
        # bridge back out the same profile if not specified
        # (the default action taken by bridge)
        profile=out_profile,
        gateway=gateway,
        # always use the SIP Request-URI
        dest_url=sess['variable_sip_req_uri'],
        proxy=proxy,
    )
```

Which defines that:

- all international calls will be blocked.
- any *inbound* calls prefixed with `1` will be *bridged* to our long distance provider.
- all `2xxx` dialed numbers will be directed to the sales PBX.
- all `4xxx` dialed numbers will be directed to the support PBX.

Notice that we can *parameterize* the inputs to the routing function using kwargs. This lets you specify data inputs you'd like used when a particular field matches. If not provided, sensible defaults can be specified in the function signature.

Also note that the idea of transferring to a context becomes a simple function call:

```
@router.route("^(XXXxxxxxxx)$")
def test_did(sess, match, router):
    # call our route function from above
    return bridge2dest(sess, match, router, profile='external')
```

Just as before, we can run our `router` as a service and use a single "dialplan" for all nodes in our *FreeSWITCH* cluster:

```
s = Service(['FS_host1.com', 'FS_host2.com', 'FS_host3.com'])
s.apps.load_app(router, app_id='default')
s.run()  # blocks forever
```

Note: If you'd like to try out *switchy* routes alongside your existing XML dialplan (assuming you've added the *park only* context in your existing config) you can either pass in `{"Caller-Context": "switchy"}` as a `guard` or you can load the router with:

```
s.apps.load_app(router, app_id='switchy', header='Caller-Context')
```

Replicating XML dialplan features

The main difference with using *switchy* for call control is that everything is processed at **runtime** as opposed to having separate *parse* and *execute* phases.

Retrieving Variables

Accessing variable values from *FreeSWITCH* is already built into *switchy*'s Session API using traditional getitem access.

Basic Logic

As a first note, you can accomplish any "logical" *field* pattern match either directly in Python or by the *regex* expression to `Router.route`:

Here is the equivalent of the logical AND example:

```python
from datetime import datetime

@router.route('^500$')
def on_sunday(sess, match, router, profile='internal', did='500'):
    """On Sunday no one works in support...
    """
    did = '531' if datetime.today().weekday() == 6 else did
    sess.bridge('{}@example.com'.format(did), profile=profile)
```

And the same for logical OR example:

```python
import re

# by regex
@router.route('^500$|^502$')
def either_ext(sess, match, router):
    sess.answer()
    sess.playback('ivr/ivr-welcome_to_freeswitch.wav')

# by if statement
@router.route('^.*$')
def match(sess, match, router):
    if re.match("^Michael\s*S?\s*Collins", sess['variable_caller_id_name']) or\
            re.match("^1001|3757|2816$", sess['variable_caller_id_number']):
        sess.playback("ivr/ivr-dude_you_rock.wav")
    else:
        sess.playback("ivr/ivr-dude_you_suck.wav")
```

Nesting logic

Nested conditions Can be easily accomplished using plain old if statements:

```
@router.route('^1.*(\d)$')
def play_wavfile(sess, match, router):
    # get the last digit
    last_digit = match.groups()[0]

    # only play the extra file when last digit is '3'
    if last_digit == '3':
        sess.playback('foo.wav')

    # always played if the first digit is '1'
    sess.playback('bar.wav')
```

Break on true

Halting all further route execution (known as break on true) can be done by raising a special error:

```
@router.route('^1.*(\d)$')
def play_wavfile(sess, match, router):
    sess.playback('foo.wav')

    if not sess['Caller-Destination-Number'] == "1100":
        raise router.StopRouting  # stop all further routing
```

Record a random sampling of call center agents

Here's an example of randomly recording call-center agents who block their outbound CID:

```
import random

@router.route('^\*67(\d+)$')
def block_cid(sess, match, router):
    did = match.groups()[0]

    if sess.is_outbound():
        # mask CID
        sess.broadcast('privacy::full')
        sess.setvars({'privacy': 'yes', 'sip_h_Privacy': 'id'})

        if random.randint(1, 6) == 4:
            sess.log.debug("recording a sneaky agent to /tmp/agents/")
            sess.start_record('/tmp/agents/{}_to_{}.wav'.format(sess.uuid, did))
```

4.14 Call generation and stress testing

Switchy contains a built in auto-dialer which enables you to drive multiple *FreeSWITCH* processes as a call generator cluster.

Once you have a set of servers *deployed*, have started *FreeSWITCH* processes on each **and** have configured *ESL* to listen on the default *8021* port, simply load the originator *app* passing in a sequence of slave server host names:

```
>>> from switchy import get_originator
>>> originator = get_originator(['hostnameA', 'hostnameB', 'hostnameC'])
>>> originator
```

```
<Originator: '0' active calls, state=[INITIAL], rate=30 limit=1
max_sessions=inf duration=10.03>
```

Note: If using ESL ports different then the default *8021*, simply pass a sequence of *(host, port)* socket pairs to the `get_originator` factory.

Now we have a binding to an `Originator` instance which is a non-blocking Switchy application allowing us to originate calls from our *FreeSWITCH* cluster.

Notice the load settings such as *rate*, *limit* and *duration* shown in the output of the originator's `__repr__()` method. These parameters determine the type of traffic which will be originated from the cluster to your target software under test (*SUT*) and downstream *callee* systems.

In order to ensure that calls are made successfully it is recommended that the *SUT* system *loop calls back* to the originating server's *caller*. This allows switchy to associate *outbound* and *inbound* SIP sessions into calls. As an example if the called system is another FreeSWITCH server under test then you can configure a *proxy dialplan*.

4.14.1 A single call generator

For simplicity's sake let's assume for now that we only wish to use **one** *FreeSWITCH* process as a call generator. This simplifies the following steps which otherwise require the more advanced `switchy.distribute` module's cluster helper components for orchestration and config of call routing. That is, assume for now we only passed *'vm-host'* to the originator factory function above.

To ensure all systems in your test environment are configured correctly try launching a single call (by keeping *limit=1*) and verify that it connects and stays active:

```
>>> originator.start()
Feb 24 12:59:14 [ERROR] switchy.Originator@['vm-host'] call_gen.py:363 : 'MainProcess' failed with:
Traceback (most recent call last):
  File "sangoma/switchy/apps/call_gen.py", line 333, in _serve_forever
      "you must first set an originate command")
ConfigurationError: you must first set an originate command
```

Before we can start generating calls we must set the command which will be used by the application when instructing each slave to *originate* a call.

Note: The error above was not raised as a Python exception but instead just printed to the screen to avoid terminating the event processing loop in the `switchy.observe.EventListener`.

Let's set an originate command which will call our *SUT* as it's first hop with a destination of *ourselves* using the default *external* profile and the *FreeSWITCH* built in *park* application for the outbound session's post-connect execution:

```
>>> originator.pool.clients[0].set_orig_cmd(
    dest_url='doggy@hostnameA:5080,
    profile='external',
    app_name='park',
    proxy='doggy@intermediary_hostname:5060',
)
>>> originator.originate_cmd  # show the rendered command str
['originate {{originator_codec=PCMU,switchy_client={app_id},
originate_caller_id_name=Mr_Switchy,originate_timeout=60,absolute_codec_string=,
sip_h_X-originating_session_uuid={uuid_str},sip_h_X-switchy_client={app_id},
origination_uuid={uuid_str}}}sofia/external/doggy@hostnameA:5060;
fs_path=sip:goodboy@intermediary_hostname:5060 &park()']
```

The underlying originate command has now been set for the **first** client in the *Orignator* app's client pool. You might notice that the command is a `format` string which has some placeholder variables set. It is the job of the `switchy.observe.Client` to fill in these values at runtime (i.e. when the `switchy.observe.Client.originate()` is called). For more info on the *originate* cmd wrapper see `build_originate_cmd()`. Also see the Internals tutorial.

Try starting again:

```
>>> originator.start()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "switchy/apps/call_gen.py", line 479, in start
    raise utils.ConfigurationError("No apps have been loaded")
switchy.utils.ConfigurationError: No apps have been loaded
```

We need to explicitly load a switchy app which will be used to process originated (and possibly received) calls. For stress testing the `switchy.apps.bert.Bert` app is recommended as it performs a stringent audio check alongside a traditional call flow using mod_bert:

```
>>> from switchy.apps.bert import Bert
>>> originator.load_app(Bert)
```

Note: The *Originator* actually supports loading multiple (groups of) apps with different *weights* such that you can execute multiple call flows in parallel. This can be useful for simulating auto-dialer traffic:

```
>>> from switchy.apps.blockers import CalleeRingback, CalleeBlockOnInvite
>>> originator.load_app(CalleeRingback, ppkwargs={'caller_hup_after': 5, 'ring_response': 'ring_ready
>>> originator.load_app(CalleeBlockonInvite, ppkwargs={'response': 404}, weight=33)
>>> originator.load_app(Bert, weight=34)
```

Try starting once more:

```
>>> originator.start()
Feb 24 14:12:35 [INFO] switchy.Originator@['vm-host'] call_gen.py:395 : starting loop thread
Feb 24 14:12:35 [INFO] switchy.Originator@['vm-host'] call_gen.py:376 : State Change: 'INITIAL' -> '(
```

At this point there should be one active call from your *caller* (bridged) through the *SUT* and then received by the *callee*. You can check the `Originator` status via it's `__repr__()` again:

```
>>> originator
<Originator: '1' active calls, state=[ORIGINATING], rate=30 limit=1 max_sessions=inf duration=10.0333
```

Warning: If you start seeing immediate errors such as:

```
Feb 24 14:12:35 [ERROR] switchy.EventListener@vm-host observe.py:730 : Job '16f6313e-bc59-11e4-8b27
-ERR NORMAL_TEMPORARY_FAILURE
```

it may mean your *callee* isn't configured correctly. Stop the *Originator* and Check the *FreeSWITCH* slave's logs to debug.

The *Originator* will keep offering new calls indefinitely with *duration* seconds allowing up to *limit*'s (in *erlangs*) worth of concurrent calls until stopped. That is, continuous load is offered until you either *stop* or *hupall* calls. You can verify this by ssh-ing to the slave and calling the *status* command from fs_cli.

You can now increase the call load parameters:

```
>>> originator.rate = 50  # increase the call rate
>>> originator.limit = 1000  # increase max concurrent call limit (erlangs)
# wait approx. 3 seconds
>>> originator
<Originator: '148' active calls, state=[INITIAL], rate=50 limit=1000 max_sessions=inf duration=30.0>
```

Note how the *duration* attribute was changed automatically. This is because the *Originator* computes the correct *average call-holding time* by the most basic erlang formula. Feel free to modify the load parameters in real-time as you please to suit your load test requirements.

To tear down calls you can use one of `stop()` or `hupall()`. The former will simply stop the *burst* loop and let calls slowly teardown as per the *duration* attr whereas the latter will forcefully abort all calls associated with a given *Client*:

```
>>> originator.hupall()
Feb 24 16:37:16 [WARNING] switchy.Originator@['vm-host'] call_gen.py:425 : Stopping all calls with hu
Feb 24 16:37:16 [INFO] switchy.Originator@['vm-host'] call_gen.py:376 : State Change: 'ORIGINATING' -
Feb 24 16:37:16 [INFO] switchy.Originator@['vm-host'] call_gen.py:357 : stopping burst loop...
Feb 24 16:37:16 [INFO] switchy.Originator@['vm-host'] call_gen.py:326 : Waiting for start command...
Feb 24 16:37:16 [ERROR] switchy.EventListener@vm-host observe.py:730 : Job '4d8823c4-bc6d-11e4-af92-1
-ERR NORMAL_CLEARING
Feb 24 16:37:16 [ERROR] switchy.EventListener@vm-host observe.py:730 : Job '4d8f509a-bc6d-11e4-afa3-1
-ERR NORMAL_CLEARING
Feb 24 16:37:16 [INFO] switchy.Originator@['vm-host'] call_gen.py:231 : all sessions have ended...
```

When *hupall*-ing, a couple *NORMAL_CLEARING* errors are totally normal.

4.14.2 Slave cluster

In order to deploy call generation clusters some slightly more advanced configuration steps are required to properly provision the `switchy.apps.call_gen.Originator`. As mentioned previous, this involves use of handy cluster helper components provided with Switchy.

The main trick is to configure each `switchy.observe.Client` to have the appropriate originate command set such that calls are routed to where you expect. A clever and succint way to accomplish this is by using the `switchy.distribute.SlavePool`. Luckily the *Originator* app is built with one internally by default.

Configuration can now be done with something like:

```
originator.pool.evals(
    ("""client.set_orig_cmd('park@{}:5080'.format(client.server),
     app_name='park',
     proxy='doggy@{}:5060'.format(ip_addr))"""),
     ip_addr='intermediary_hostname.some.domain'
)
```

This will result in each slave calling itself *through* the intermediary system. The *pool.evals* method essentially allows you to invoke arbitrary Python expressions across all slaves in the cluster.

For more details see *Cluster tooling* .

4.14.3 Measurement collection

By default, the *Originator* collects call detail records using the built-in *CDR* app. Given that you have pandas installed this data and additional stress testing metrics can be accessed in *pandas* DataFrames via the `switchy.apps.call_gen.Originator.measurers` object:

```
>>> orig.measurers.stores.CDR
      switchy_app  hangup_cause     caller_create  caller_answer caller_req_originate  caller_origina
0     Bert         NORMAL_CLEARING  1.463601e+09   1.463601e+09  1.463601e+09          1.463601e+09
1     Bert         NORMAL_CLEARING  1.463601e+09   1.463601e+09  1.463601e+09          1.463601e+09
2     Bert         NORMAL_CLEARING  1.463601e+09   1.463601e+09  1.463601e+09          1.463601e+09
3     Bert         NORMAL_CLEARING  1.463601e+09   1.463601e+09  1.463601e+09          1.463601e+09
...
1056  Bert         NORMAL_CLEARING  1.463601e+09   1.463601e+09  1.463601e+09          1.463601e+09

>>> originator.measurers.ops.call_metrics
       active_sessions  answer_latency  avg_call_rate  call_duration \
0      8                0.020000        NaN             20.880000
1      12               0.020000        NaN             20.820000
2      22               0.020000        NaN             20.660000
3      2                0.020000        NaN             20.980000
...

       call_rate  call_setup_latency  erlangs  failed_calls  \
0      25.000024  0.060000            4        0
1      49.999452  0.060000            6        0
2      50.000048  0.060000            11       0
3      NaN        0.120000            1        0
...
```

If you have matplotlib installed you can also plot the results using `Originator.measurers.plot()`.

If you do not have have *pandas* installed then the CDR records are still stored in a local *csv* file and can be read into a list of lists using the same `orig.measurers.stores.CDR` attribute.

More to come...

4.15 Command line

Switchy provides a convenient cli to initiate load tests with the help of click. The program is installed as binary *switchy*:

```
$ switchy
Usage: switchy [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  list-apps
  plot
  run
```

A few sub-commands are provided. For example you can list the applications available (Call Applications determine call flows):

```
$ switchy list-apps
Collected 5 built-in apps from 7 modules:

switchy.apps.bert:

`Bert`: Call application which runs the bert test application on both legs of a call
```

```
    See the docs for `mod_bert`_ and discussion by the author `here`_.

    .. _mod_bert:
        https://freeswitch.org/confluence/display/FREESWITCH/mod_bert
    .. _here:
        https://github.com/moises-silva/freeswitch/issues/1

switchy.apps.players:

`TonePlay`: Play a 'milli-watt' tone on the outbound leg and echo it back on the inbound

`PlayRec`: Play a recording to the callee and record it onto the local file system

    This app can be used in tandem with MOS scoring to verify audio quality.
    The filename provided must exist in the FreeSWITCH sounds directory such that
    ${FS_CONFIG_ROOT}/${sound_prefix}/<category>/<filename> points to a valid wave file.

switchy.apps.dtmf:

`DtmfChecker`: Play dtmf tones as defined by the iterable attr `sequence` with tone `duration`.
    Verify the rx sequence matches what was transmitted.  For each session which is answered start
    a sequence check. For any session that fails digit matching store it locally in the `failed` attr

switchy.apps.routers:

`Bridger`: Bridge sessions within a call an arbitrary number of times.
```

The applications listed can be used with the *app* option to the *run* sub-command. *run* is the main sub-command used to start a load test. Here is the help:

```
$ switchy run --help
Usage: switchy run [OPTIONS] SLAVES...

Options:
  --proxy TEXT                    Hostname or IP address of the proxy device
                                  (this is usually the device you are testing)
                                  [required]
  --profile TEXT                  Profile to use for outbound calls in the
                                  load slaves
  --rate TEXT                     Call rate
  --limit TEXT                    Maximum number of concurrent calls
  --max-offered TEXT              Maximum number of calls to place before
                                  stopping the program
  --duration TEXT                 Duration of calls in seconds
  --interactive / --non-interactive
                                  Whether to jump into an interactive session
                                  after setting up the call originator
  --debug / --no-debug            Whether to enable debugging
  --app TEXT                      Switchy application to execute (see list-
                                  apps command to list available apps)
  --metrics-file TEXT             Store metrics at the given file location
  --help                          Show this message and exit.
```

The *SLAVES* argument can be one or more IP's or hostnames for each configured FreeSWITCH process used to originate traffic. The *proxy* option is required and must be the IP address or hostname of the device you are testing. All slaves will direct traffic to the specified proxy.

The other options are not strictly required but typically you will want to at least specify a given call rate using the *rate* option, max number of concurrent calls (erlangs) with *limit* and possibly max number of calls offered with *max-offered*.

For example, to start a test using an slave located at *1.1.1.1* to test device at *2.2.2.2* with a maximum of *2000* calls at *30* calls per second and stopping after placing *100,000* calls you can do:

```
$ switchy run 1.1.1.1 --profile external --proxy 2.2.2.2 --rate 30 --limit 2000 --max-offered 100000

Slave 1.1.1.1 SIP address is at 1.1.1.1:5080
Starting load test for server 2.2.2.2 at 30cps using 1 slaves
...
```

Note that the *profile* option is also important and the profile must exist already for all specified slaves.

In this case the call duration would be automatically calculated to sustain that call rate and that max calls exactly, but you can tweak the call duration in seconds using the *duration* option.

Additionally you can use the *metrics-file* option to store call metrics in a file. You can then use the *plot* sub-command to generate graphs of the collected data using *matplotlib* if installed.

4.16 Session API

switchy wraps *FreeSWITCH*'s event header fields and call management commands inside the *switchy.models.Session* type.

There is already slew of supported commands and we encourage you to add any more you might require via a pull request on github.

4.16.1 Accessing *FreeSWITCH* variables

Every `Session` instance has access to all it's latest received *event headers* via standard python `__getitem__` access:

```
sess['Caller-Direction']
```

All chronological event data is kept until a `Session` is destroyed. If you'd like to access older state you can use the underlying *Events* instance:

```
# access the first value of my_var
sess.events[-1]['variable_my_var']
```

Note that there are some distinctions to be made between different types of variable access and in particular it would seem that *FreeSWITCH*'s event headers follow the info app names:

```
# standard headers require no prefix
sess['FreeSWITCH-IPv6']
sess['Channel-State']
sess['Unique-ID']

# channel variables require a 'variable_' prefix
sess['variable_sip_req_uri']
sess['variable_sip_contact_user']
sess['variable_read_codec']
sess['sip_h_X-switchy_app']
```

4.17 Internals tutorial

Getting familiar with Switchy's guts means learning to put the appropriate components together to generate a call. This simple guide is meant to provide some commentary surrounding low level components and interfaces so that you can begin reading the source code. It is assumed you are already familiar with the prerequisite deployment steps.

4.17.1 Primary Components

Currently there are 3 main objects in Switchy for driving *FreeSWITCH*:

`Connection` - a thread safe wrapper around the ESL SWIG python package's *ESLConnection*

`EventListener` - the type that contains the core event processing loop and logic

- Primarily concerned with observing and tracking the state of a single *FreeSWITCH* process
- Normally a one-to-one pairing of listeners to slave processes/servers is recommended to ensure deterministic control.
- Contains a `Connection` used mostly for receiving events only transmitting ESL commands when dictated by Switchy apps

`Client` - a client for controlling *FreeSWITCH* using the ESL inbound method

- contains a `Connection` for direct synchronous commands and optionally an `EventListener` for processing asynchronous calls

For this guide we will focus mostly on the latter two since they are the primary higher level components the rest of the library builds upon.

4.17.2 Using a *Client* and *EventListener* pair

A Client can be used for invoking or sending **synchronous** commands to the *FreeSWITCH* process. It handles ESL *api* calls entirely on it's own.

To connect simply pass the hostname or ip address of the slave server at instantiation:

```
>>> from switchy import Client
>>> client = Client('vm-host')
>>> client.connect()  # could have passed the hostname here as well
>>> client.api('status')  # call ESL `api` command directly
<ESL.ESLevent; proxy of <Swig Object of type 'ESLevent *' at 0x28c1d10> >

>>> client.cmd('global_getvar local_ip_v4')  # `api` wrapper which returns event body content
'10.10.8.21'

>>> client.cmd('not a real command')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "switchy/observe.py", line 1093, in cmd
      return self.api(cmd).getBody().strip()
    File "switchy/observe.py", line 1084, in api
      consumed, response = EventListener._handle_socket_data(event)
    File "switchy/observe.py", line 651, in _handle_socket_data
      raise APIError(body)
switchy.utils.APIError: -ERR not Command not found!
```

Now let's initiate a call originating from the slave process's *caller* which is by default the external sip profile:

```
>>> client.originate(dest_url='9196@intermediary_hostname:5060')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "switchy/observe.py", line 1177, in originate
        listener = self._assert_alive(listener)
    File "switchy/observe.py", line 1115, in _assert_alive
        assert self.listener, "No listener associated with this client"
    File "switchy/observe.py", line 973, in get_listener
        "No listener has been assigned for this client")
    AttributeError: No listener has been assigned for this client
```

The *Client* implements *originate* by making an **asynchronous** ESL *bgapi* call to the slave process. In order to track the eventual results of that call, an *EventListener* must be used which will collect the state changes triggered by the command (i.e. as received in event data from the slave process).

With this current architecture you can think of a *listener* as an object from which you can track *FreeSWITCH* state and a *client* as an interface which drives the slave process using commands to trigger **new** state(s). Again, any time a *Client* makes an **asynchronous** call an *EventListener* is needed to handle and report back the result(s).

Let's create and assign an `EventListener`:

```
>>> from switchy import get_listener
>>> l = get_listener('vm-host')
>>> l  # initially disconnected to allow for unsubcriptions from the default event set
<EventListener [disconnected]>
>>> l.connect()
Feb 25 10:33:05 [INFO] switchy.EventListener@vm-host observe.py:346 : Connected listener 'd2d4ee82-bc
>>> l
<EventListener [connected]>
>>> l.start()
Feb 25 10:35:30 [INFO] switchy.EventListener@vm-host observe.py:287 : starting event loop thread
>>> client.listener = l
```

Note: Alternatively an *EventListener* can be passed to the *Client* at instatiation time.

Now let's attempt our *originate* once more this time executing the *9197* extension once the *caller* is answered, and calling the *echo* extension, *9196*, at the *callee* end:

```
>>> client.originate('9196@vm-host:5080',
    dp_exten=9197,
    proxy='intermediary_hostname:5060'
)
<switchy.models.Job at 0x7feea01c6c90>

>>> client.listener.calls  # check the active calls collection
OrderedDict([('72451178-bd0c-11e4-9d26-74d02bc595d7', <Call(72451178-bd0c-11e4-9d26-74d02bc595d7, 2 s
```

Note: See the *default* dialplan packaged with stock *FreeSWITCH*. Use of these extensions assumes you have assigned the external sip profile to use the *default* dialplan by assigning it's *context* parameter

The async *originate* call returns to us a `switchy.models.Job` instance (as would any call to `switchy.observe.Client.bgapi()`). A *Job* provides the same interface as that of the `multiprocessing.pool.AsyncResult` and can be handled to completion synchronously:

```
>>> job = client.originate('9196@vm-host:5080',
    dp_exten=9197,
    proxy='intermediary_hostname:5060
)
>>> job.get(timeout=30)  # block up to 30 seconds waiting for result
'4d9b4128-bd0f-11e4-9d26-74d02bc595d7'  # the originated session uuid

>>> job.sess_uuid   # a special attr which is always reserved for originate results (i.e. session ids
'4d9b4128-bd0f-11e4-9d26-74d02bc595d7'

>>> client.hupall()  # hangup the call
```

4.17.3 Call control using Switchy apps

To use Switchy at its fullest potential, applications can be written to process state tracked by the *EventListener*. The main benefit is that apps can be written in pure Python somewhat like the mod_python module provided with *FreeSWITCH*. Switchy gives the added benefit that the Python process does not have to run on the slave machine and in fact **multiple** applications can be managed independently of **multiple** slave configurations thanks to Switchy's use of the ESL inbound method.

App Loading

Switchy apps are loaded using `switchy.observe.Client.load_app()`. Each app is referenced by it's appropriate name (if none is provided) which allows for the appropriate callback lookups to be completed by the *EventListener*.

We can now accomplish the same tone play steps from above using the built-in *TonePlay* app:

```
>>> from switchy.apps.players import TonePlay
>>> client.load_app(TonePlay)
Feb 25 13:27:43 [INFO] switchy.Client@vm-host observe.py:1020 : Loading call app 'TonePlay'
'fd27be58-bd1b-11e4-b22d-74d02bc595d7'  # the app uuid since None provided

>>> client.apps.TonePlay
<switchy.apps.players.TonePlay at 0x7f7c5fdaf650>

>>> isinstance(client.apps.TonePlay, TonePlay)  # Loading the app type instantiates it
True
```

Note: App loading is *atomic* so if you mess up app implementation you don't have to worry that inserted callbacks are left registered with the *EventListener*

Assuming the Switchy *park-only dialplan* is used by the external sip profile we can now originate our call again:

```
>>> job = client.originate('park@vm-host:5080',
    proxy='intermediary_hostname:5060',
    app_id=client.apps.TonePlay.cid
)
>>> job.wait(10)  # wait for call to connect
>>> call = client.listener.calls[job.sess_uuid]  # look up the call by originating sess uuid
>>> call.hangup()
```

4.17.4 Example Snippet

As a summary, here is an snippet showing all these steps together:

```python
import time
from switchy import Client, EventListener
from switchy.apps.players import TonePlay

# init
listener = EventListener('vm-host')
client = Client('vm-host', listener=listener)
client.connect()
listener.connect()
listener.start()

# app load
id = client.load_app(TonePlay)
# make a call
job = client.originate(
    dest_url='park@vm-host',
    proxy='intermediary_hostname',
    app_id=id
)
sessid = job.get(30)
assert sessid == job.sess_uuid
# hangup
call = client.listener.calls[job.sess_uuid]
orig_sess = call.sessions[0]  # get the originating session
time.sleep(10)  # let it play a bit
orig_sess.hangup()
```

Conveniently enough, the boilerplate here is almost exactly what the `active_client()` context manager does internally. An example of usage can be found in the quickstart guide.

4.18 Running Unit Tests

Switchy's unit test set relies on pytest and tox. Tests require a *FreeSWITCH* slave process which has been deployed with the required baseline config and can be accessed by hostname.

To run all tests invoke *tox* from the source dir and pass the FS hostname:

```
tox -e ALL -- --fshost=hostname.fs.com
```

SIPp and pysipp are required to be installed locally in order to run call/load tests.

To run multi-slave tests at least two slave hostnames are required:

```
tox -e ALL -- --fsslaves='["fs.slave.hostname1","fs.slave.hostname2"]'
```

Index