

Contents

- 1 Introduction
 - ◆ 1.1 Inbound
 - ◆ 1.2 Outbound
- 2 Configuration
 - ◆ 2.1 IPv6
 - ◆ 2.2 ACL
- 3 Example Clients
 - ◆ 3.1 Liverpie
 - ◆ 3.2 Telnet Client
 - ◆ 3.3 Perl Command Client
 - ◆ 3.4 Ruby Libraries
 - ◆ 3.5 Python Client Library
 - ◆ 3.6 PHP Client
 - ◆ 3.7 Java Client Libraries
 - ◆ 3.8 Javascript / Node.js library
 - ◆ 3.9 .NET Client library
- 4 Command Documentation
 - ◆ 4.1 api
 - ◆ 4.2 bgapi
 - ◆ 4.3 linger
 - ◆ 4.4 nolinger
 - ◆ 4.5 event
 - ◇ 4.5.1 Special Case - 'myevents'
 - ◇ 4.5.2 divert_events
 - ◇ 4.5.3 Read more
 - ◆ 4.6 filter
 - ◆ 4.7 filter delete
 - ◆ 4.8 sendevent
 - ◆ 4.9 sendmsg
 - ◇ 4.9.1 call-command
 - 4.9.1.1 execute
 - 4.9.1.2 hangup
 - 4.9.1.3 unicast
 - 4.9.1.4 nomedia

- ◆ [4.10 exit](#)
- ◆ [4.11 auth](#)
- ◆ [4.12 log](#)
- ◆ [4.13 nolog](#)
- ◆ [4.14 nixevent](#)
- ◆ [4.15 noevents](#)
- [5 Sample Event Socket Applications](#)
- [6 See Also](#)

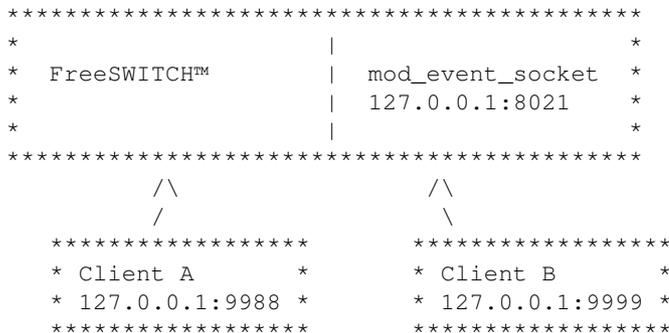
Introduction

mod_event_socket is a TCP based interface to control FreeSWITCH. The default values are to bind to 127.0.0.1 port 8021 and the default password is ClueCon. See the Configuration section below for a note on when to use 0.0.0.0 instead of 127.0.0.1.

The module operates in two modes, inbound and outbound.

Inbound

Inbound mode means you run your applications (in whatever languages) as clients and connect to the FreeSWITCH server to invoke commands and control FreeSWITCH.



In inbound mode, your application (Client A, Client B: Python, Perl, etc) connects to the FreeSWITCH™ server on the given port and sends commands, as shown in the above diagram. The rest of this document is biased toward this inbound mode, though there is a lot of overlap with outbound mode. Using inbound socket connections you can check status, make outbound calls, etc.

Outbound

Outbound mode means you make a daemon (with whatever language) and then have FS connect to it. You add an extension to the dialplan and put `<action application="socket" data="ip:port sync full"/>` and create a script that runs on that ip:port and answer, playback and everything else you need on the script. Since revision git-8c794ac on 14/03/2012 you can connect to IPv6 addresses. When using IPv6 addresses the port parameter is required: `<action application="socket" data="::1:8021"/>` connects to ::1 on port 8021. Since this revision hostnames resolving to IPv6 addresses can be used.

Mod_event_socket

In outbound mode, also known as the "socket application" (or socket client), FreeSWITCH™ makes outbound connections to another process (similar to Asterisk's Fagi model). Using outbound connections you can have FreeSWITCH™ call your own application(s) when particular events occur. See [Event Socket Outbound](#) for more details regarding things specific to outbound mode.

Configuration

First enable the mod_event_socket module in modules.conf.xml (in a default configuration this is located in /usr/local/freeswitch/conf/autoload_configs/modules.conf.xml)

By default the module is enabled but restricted to localhost by the settings specified below.

The module settings can be changed in the event_socket.conf.xml file (in a default configuration this is located in /usr/local/freeswitch/conf/autoload_configs/event_socket.conf.xml):

```
<configuration name="event_socket.conf" description="Socket Client">
  <settings>
    <param name="listen-ip" value="127.0.0.1"/>
    <param name="listen-port" value="8021"/>
    <param name="password" value="ClueCon"/>
  </settings>
</configuration>
```

The default settings allow socket connections only from the local host. To allow connections from any host on the network, use 0.0.0.0 instead of the default 127.0.0.1:

```
<configuration name="event_socket.conf" description="Socket Client">
  <settings>
    <!-- Allow socket connections from any host -->
    <param name="listen-ip" value="0.0.0.0"/>
    <param name="listen-port" value="8021"/>
    <param name="password" value="ClueCon"/>
  </settings>
</configuration>
```

IPv6

Since revision git-8c794ac on 14/03/2012 you can listen on IPv6 addresses. On dual stack hosts you can listen on both IPv4 and IPv6 addresses using:

```
<configuration name="event_socket.conf" description="Socket Client">
  <settings>
    <!-- Allow socket connections from any host -->
    <param name="listen-ip" value="::"/>
    <param name="listen-port" value="8021"/>
    <param name="password" value="ClueCon"/>
  </settings>
</configuration>
```

ACL

Access lists can be created in [acl.conf.xml](#) and enabled in [event_socket.conf.xml](#) or allow ip ranges directly in [event_socket.conf.xml](#)

```
<param name="apply-inbound-acl" value="<acl_list|cidr"/>
```

Example:

```
<param name="apply-inbound-acl" value="10.20.0.0/16"/>
```

Note that multiple apply-inbound-acl params will not work.

Example Clients

The idea is that you would write your own client, but here are some examples.

Liverpie

Liverpie (language independent IVR proxy) is a free little piece of software, written in Ruby, that talks to FreeSWITCH on one side, and to any web application on the other, regardless of language, platform etc. It translates FreeSWITCH mod_socket dialogue into HTTP talk (embedding various parameters in HTTP headers), so you can write your own http-speaking finite state machine and hook it to FreeSWITCH via Liverpie. Note also that Liverpie expects the response in YAML so you can save yourself the pain of providing XML if you are comfortable with Liverpie doing the translation.

Find Liverpie here: www.liverpie.com

Telnet Client

telnet to port 8021 and enter "auth ClueCon" as password to authenticate. From here you can send any of the commands listed in this document. Note that you need to provide a double empty line after the command so it processed.

Perl Command Client

There is a Perl client in scripts/socket called fs.pl.

With the module up and loaded:

```
cd scripts/socket
perl fs.pl <optional log level>
```

You can enter a few api commands like "show" or "status".

Ruby Libraries

- [FreeSWITCHeR](#)
- [Librevox](#)
- [RubyFS](#)

Python Client Library

In the scripts/python/freepy directory, there is a python client library based on the [Twisted](#) asynchronous socket library. An example is included.

There's also an alternative [twisted protocol class for FreeSWITCH's event socket](#). It currently supports inbound and outbound methods, and examples are included.

[PySWITCH](#) is one more library for Python and Twisted programmers. It has extensive support for FreeSWITCH API, bgapi and Dialplan Tools.

PHP Client

Website based PHP example [PHP Event Socket](#)

Java Client Libraries

[Java ESL](#) describes several options for using Java to communicate with the ESL.

Javascript / Node.js library

- Node-esl module available on [GitHub](#) and on [npm](#). Provides support for both Inbound and Outbound connections, a Server helper for multiple Outbound connections, and implements the [Event Socket Library](#) interface. Examples can be found in the [examples directory](#) of the source.
- Node.js esl module ([available using npm](#)) offers both a client and a server implementation. The code is [on github](#) with [documentation](#). Examples: [Voicemail with CouchDB storage](#) and [CNAM injection](#) (short example showing how to set a variable using an async web query).
- [Node.js ESClient](#)
- Possible example of an IVR:
<http://freeswitch-users.2379917.n2.nabble.com/Javascript-Outbound-Event-Socket-Linger-Command-td75309>

.NET Client library

Support is included in tree for building under Windows. See `libs/esl/managed managed_esl.201x`.

There is a sample application included named `ManagedEslTest`. This will demonstrate how to start a simple inbound example as provided but also shows the other modes as well.

The managed portions are platform neutral but the `esl swig wrapper` (the `ESL` project in the managed solution) is platform dependent and must be built for the intended platform. You must also have previously built the main `FreeSWITCH` project as the native `esl` library is included from there otherwise you will receive a `esl.lib not found` error.

[\[1\]](#) AgbaraVOIP .NET Client

Command Documentation

The following section aims at documenting all commands that can be sent. This section is work in progress.

api

Send an `api` command (blocking mode)

```
api <command> <arg>
```

`Api` commands are documented in the [mod_commands](#) section.

Examples:

```
api originate sofia/mydomain.com/ext@yourvsp.com 1000 # connect sip:ext@yourvsp.com to extension
api sleep 5000
```

bgapi

Send an `api` command (non-blocking mode) this will let you execute a job in the background and the result will be sent as an event with an indicated `uuid` to match the reply to the command)

```
bgapi <command> <arg>
```

The same `API` commands available as with the **api** command, however the server returns immediately and is available for processing more commands.

Example return value:

```
Content-Type: command/reply
Reply-Text: +OK Job-UUID: c7709e9c-1517-11dc-842a-d3a3942d3d63
```

Mod_event_socket

When the command is done executing, freeswitch fires an event with the result and you can compare that to the Job-UUID to see what the result was. In order to receive this event, you will need to subscribe to BACKGROUND_JOB events.

If you want to set your own custom Job-UUID over plain socket:

```
bgapi status
Job-UUID: d8c7f660-37a6-4e73-9170-1a731c442148
```

Reply:

```
Content-Type: command/reply
Reply-Text: +OK Job-UUID: d8c7f660-37a6-4e73-9170-1a731c442148
Job-UUID: d8c7f660-37a6-4e73-9170-1a731c442148
```

linger

Tells FreeSWITCH not to close the socket connect when a channel hangs up. Instead, it keeps the socket connection open until the last event related to the channel has been received by the socket client.

```
linger
```

nolinger

Disable socket lingering. See **linger** above.

```
nolinger
```

event

Enable or disable events by class or all (plain or xml or json output format)

```
event plain <list of events to log or all for all>
```

The event command are used to subscribe on events from FreeSWITCH. You may specify any number events on the same line, they should be separated with space.

The events are listed in the [Event List](#) page.

Examples:

```
event plain ALL
event plain CHANNEL_CREATE CHANNEL_DESTROY CUSTOM conference::maintenance sofia::register sofia
event xml ALL
event json CHANNEL_ANSWER
```

Subsequent calls to 'event' won't override the previous event sets. Supposing, you've first registered for DTMF

Mod_event_socket

```
event plain DTMF
```

then you may want to register for CHANNEL_ANSWER also, it is enough to give

```
event plain CHANNEL_ANSWER
```

and you will continue to receive DTMF along with CHANNEL_ANSWER, later one doesn't override the previous.

Special Case - 'myevents'

The 'myevents' subscription allows your inbound socket connection to behave like an outbound socket connect. It will "lock on" to the events for a particular uuid and will ignore all other events, closing the socket when the channel goes away or closing the channel when the socket disconnects and all applications have finished executing.

Usage:

```
myevents <uuid>
```

Once the socket connection has locked on to the events for this particular uuid it will NEVER see any events that are not related to the channel, even if subsequent **event** commands are sent. If you need to monitor a specific channel/uuid *and* you need watch for other events as well then it is best to use a filter.

You can also set the event format (plain, xml or json):

Usage:

```
myevents plain <uuid>
myevents json <uuid>
myevents xml <uuid>
```

The default format is plain.

divert_events

The divert_events switch is available to allow events that an embedded script would expect to get in the inputcallback to be diverted to the event socket.

Examples:

```
divert_events on
divert_events off
```

An inputcallback can be registered in an embedded script using setInputCallback(). Setting divert_events to "on" can be used for chat messages like talk channel, ASR events and others.

Read more

- [Event List](#)

filter

Specify event types to listen for. Note, this is not a filter out but rather a "filter in," that is, when a filter is applied only the filtered values are received. Multiple filters on a socket connection are allowed.

Usage:

```
filter <EventHeader> <ValueToFilter>
```

Example:

The following example will subscribe to all events and then create two filters, one to listen for HEARTBEATS and one to listen for CHANNEL_EXECUTE events.

```
events plain all

Content-Type: command/reply
Reply-Text: +OK event listener enabled plain

filter Event-Name CHANNEL_EXECUTE

Content-Type: command/reply
Reply-Text: +OK filter added. [filter]=[Event-Name CHANNEL_EXECUTE]

filter Event-Name HEARTBEAT

Content-Type: command/reply
Reply-Text: +OK filter added. [Event-Name]=[HEARTBEAT]
```

Now only HEARTBEAT and CHANNEL_EXECUTE events will be received. You can filter on any of the event headers. To filter for a specific channel you will need to use the uuid:

```
filter Unique-ID d29a070f-40ff-43d8-8b9d-d369b2389dfe
```

This method is an alternative to the [myevents](#) event type. If you need *only* the events for a specific channel then use **myevents**, otherwise use a combination of filters to narrow down the events you wish to receive on the socket.

To filter multiple unique IDs, you can just add another filter for events for each UUID. This can be useful for example if you want to receive start/stop-talking events for multiple users on a particular conference.

```
filter plain all
filter plain CUSTOM conference::maintenance
filter Unique-ID $participantB
filter Unique-ID $participantA
filter Unique-ID $participantC
```

Mod_event_socket

This will give you events for Participant A,B and C on any conference. To receive events for all users on a conference you can use something like:

```
filter Conference-Unique-ID $ConfUUID
```

You can filter on any of the parameters you get in a freeSWITCH event:

```
filter plain all
filter call-direction Inbound
filter Event-Calling-File mod_conference.c
filter Conference-Unique-ID $ConfUUID
```

You can use them individually or compound them depending on whatever end result you desire for the type of events you want to receive

filter delete

Specify the events which you want to revoke the filter. filter delete can be used when some filters are applied wrongly or when there is no use of the filter.

Usage:

```
filter delete <EventHeader> <ValueToFilter>
```

Example:

```
filter delete Event-Name HEARTBEAT
```

Now, you will no longer receive HEARTBEAT events. You can delete any filter that is applied by this way.

```
filter delete Unique-ID d29a070f-40ff-43d8-8b9d-d369b2389dfe
```

This is to delete the filter which is applied for the given unique-id. After this, you won't receive any events for this unique-id.

```
filter delete Unique-ID
```

This deletes all the filters which are applied based on the unique-id.

sendevent

Send an event into the event system (multi line input for headers)

```
sendevent <event-name>
```

Mod_event_socket

The event generated by sendevent has the correct event type in the internal event structure, so nodes which have subscribed to this event type will get the event, but unfortunately the "Event-Name" header field is always "COMMAND". In fact, the received event is simply a clone of the original COMMAND event for "sendevent" itself.

NOTE: This apparently is a bug that is being addressed. As a work-around, you can send sendevent without specifying an event type, and include a Event-Name header with the desired event name. For example:

```
sendevent SOME_NAME
Event-Name: CUSTOM
Event-Subclass: albs::Section-Alarm
Section: 33
Alarm-Type: PIR
State: ACTIVE
```

For **MWI** you make the FreeSWITCH event SWITCH_EVENT_MESSAGE_WAITING with headers:

```
MWI-Messages-Waiting (yes/no)
MWI-Message-Account <any sip url you want>
MWI-Voice-Message x/y (a/b)
read/unread (urgent read/urgent unread)
```

To have Snom phones reread their settings from the settings server you can use:

```
sendevent NOTIFY
profile: internal
event-string: check-sync;reboot=false
user: 1000
host: 192.168.10.4
content-type: application/simple-message-summary
```

Example of sendevent with a message body, the length of the body is specified by content-length:

```
sendevent NOTIFY
profile: internal
content-type: application/simple-message-summary
event-string: check-sync
user: 1005
host: 192.168.10.4
content-length: 2
```

OK

Another example with a notify:

```
sendevent NOTIFY
profile: internal
content-type: application/simple-message-summary
event-string: check-sync
user: 1005
host: 99.157.44.194
content-length: 2
```

OK

Mod_event_socket

Results in a packet like this:

```
NOTIFY sip:1005@99.157.44.203 SIP/2.0
Via: SIP/2.0/UDP 99.157.44.194;rport;branch=z9hG4bKpH2DtBDcDtG0N
Max-Forwards: 70
From: <sip:1005@99.157.44.194>;tag=Dy3c6Q1y15v5S
To: <sip:1005@99.157.44.194>
Call-ID: 129d1446-0063-122c-15aa-001a923f6a0f
CSeq: 104766492 NOTIFY
Contact: <sip:mod_sofia@99.157.44.194:5060>
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-9578:9586
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, PRACK, MESSAGE, SUBSCRIBE, NOTIFY, REFER, UPDATE, REGISTER
Supported: 100rel, timer, precondition, path, replaces
Event: check-sync
Allow-Events: talk, presence, dialog, call-info, sla, include-session-description, presence.wininfo
Subscription-State: terminated;timeout
Content-Type: application/simple-message-summary
Content-Length: 2
```

OK

Example for Sipura/Linksys/Cisco phone or ATA to resync config with a specified profile URL:

```
sendevent NOTIFY
profile: internal
event-string: resync;profile=http://10.20.30.40/profile.xml
user: 1000
host: 10.20.30.40
content-type: application/simple-message-summary
```

Results in a packet like this:

```
NOTIFY sip:1000@10.20.30.41:5060 SIP/2.0
Via: SIP/2.0/UDP 10.20.30.40:5060;rport;branch=z9hG4bKyK4gHN28Hpyaa
Max-Forwards: 70
From: <sip:1000@10.20.30.40>;tag=FDXet6B470F6B
To: <sip:1000@10.20.30.40>
Call-ID: 19ff59fb-2cfc-1230-66b7-00199988ac0c
CSeq: 29295547 NOTIFY
Contact: <sip:mod_sofia@10.20.30.40:5060>
User-Agent: FreeSWITCH-mod_sofia/1.0.head-git-12f2bdf 2011-11-28 16-45-59 -0600
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER, NOTIFY, PUBLISH
Supported: timer, precondition, path, replaces
Event: resync;profile=http://10.20.30.40/profile.xml
Allow-Events: talk, hold, presence, dialog, line-seize, call-info, sla, include-session-description
Subscription-State: terminated;reason=timeout
Content-Type: application/simple-message-summary
Content-Length: 0
```

- If 'Auth Resync-Reboot' is set to yes (default) in the phone than you have to specify the reverse-auth-user and reverse-auth-pass fields
- If you only put 'event-string: resync' in the body than the unit will use there stored profile url

Example of send event with a MESSAGE and a message body, the length of the body is specified by content-length:

```
sendevent SEND_MESSAGE
profile: internal
content-length: 2
```

sendevent

Mod_event_socket

```
content-type: text/plain
user: 1005
host: 99.157.44.194
```

OK

Results in a packet like this:

```
MESSAGE sip:1005@99.157.44.203 SIP/2.0
Via: SIP/2.0/UDP 99.157.44.194;rport;branch=z9hG4bK0apcKrtycp64p
Max-Forwards: 70
From: <sip:1005@99.157.44.194>;tag=4c0Dp49yUNmXH
To: <sip:1005@99.157.44.194>
Call-ID: 29916da5-0062-122c-15aa-001a923f6a0f
CSeq: 104766296 MESSAGE
Contact: <sip:mod_sofia@99.157.44.194:5060>
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-9578:9586
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, PRACK, MESSAGE, SUBSCRIBE, NOTIFY, REFER, UPDATE, REGISTER
Supported: 100rel, timer, precondition, path, replaces
Content-Type: text/plain
Content-Length: 2
```

OK

To display a text message on a Snom 370 or Snom 820 the message must be of type "text/plain".

Authentication: Some phones require authentication for NOTIFY requests. FS can respond to a digest challenge if reverse authentication credentials are supplied for the user. See: [XML User Directory Guide#reverse authentication](#)

I've not found any documentation of any additional event headers, hopefully someone else will add that information. The events themselves can be found here: [Event List](#)

```
sendevent CHANNEL_HANGUP
```

sendmsg

```
sendmsg <uuid>
```

Send a message to the call of given uuid (call-command execute or hangup), see examples below.

Send message are used to send messages that controls the behavior of FreeSWITCH. You need to provide a uid (the unique id for each call/session).

originate a call directly to park by making an ext the ext part of the originate command &park()

IMPORTANT! All sendmsg commands must be followed by 2 returns.

Since messaging format is similar to RFC2882. If you are using any libraries that following line wrapping recommendation of [RFC 2822](#) make sure that you disable line wrapping as [FreeSWITCH](#) will ignore wrapped line.

```
sendmsg
```

Example:

```
sendmsg <uuid>
call-command: execute
execute-app-name: playback
execute-app-arg: /tmp/test.wav
```

call-command

The first command is Call-Command, it can do one of the following subcommands:

execute

Execute are used to execute applications, check the Dialplan XML section for more information about all commands.

The format should be:

```
sendmsg <uuid>
call-command: execute
execute-app-name: <one of the applications>
execute-app-arg: <application data>
loops: <number of times to invoke the command, default: 1>
```

This alternate format can be used for app args that are truncated by the module's 2048 octet limit:

```
sendmsg <uuid>
call-command: execute
execute-app-name: <one of the applications>
loops: <number of times to invoke the command, default: 1>
content-type: text/plain
content-length: <content length>

<application data>
```

hangup

Hangup the call.

Format:

```
sendmsg <uuid>
call-command: hangup
hangup-cause: <one of the causes listed below>
```

Additional information

- [Hangup Causes](#)

Mod_event_socket

unicast

Unicast is used to hook up spandsp for in and outbound faxing over a socket.

Additional information from Brian:

that is a nice way someone writing a script/app that uses the socket interface can get at the med
it's good because then spandsp isn't living inside of FreeSWITCH
it can run on a box sitting next to it
scales better

```
sendmsg <uuid>
call-command: unicast
local-ip: <default is 127.0.0.1>
local-port: <default is 8025>
remote-ip: <default is 127.0.0.1>
remote-port: <default is 8026>
transport: <either "tcp" or "udp", without the quotes>
and optionally
flags: native - don't transcode audio to/from L16
```

nomedia

No description.

```
sendmsg <uuid>
call-command: nomedia
nomedia-uuid: <noinfo>
```

exit

```
exit
```

Close the socket connection.

auth

```
auth <password>
```

log

```
log <level>
```

Enable log output. Levels same as the console.conf values

nolog

nolog

Disable log output previously enabled by the log command

nixevent

nixevent <event types | ALL | CUSTOM custom event sub-class>

Command to allow 'events all' followed by 'nixevent <some_event>' to do all but 1 type scenario.

noevents

noevents

Disable all events that were previously enabled with event.

Sample Event Socket Applications

- [Email2callback](#) - Email to callback application with Python and freepy.

See Also

- [Event Socket Library](#)
- [Event Socket Outbound](#)
- [Debugging Event Socket Message](#)
- [Event List](#)