mod_event_socket

0. About

`mod_event_socket` is a TCP-based interface to control FreeSWITCH, and it operates in two modes, **inbound** and **outbound**. (These terms are relative to FreeSWITCH).

By default, connections are only allowed from `localhost`, but this can be changed via configuration files (see **Configuration** section below for details).

1. Configuration

First enable the `mod_event_socket` module in `<conf_dir>/autoload_configs/modules.conf.xml` by removing the comment marks (`<!--` and `-->`).

Enabled by default

`mod_event_socket` is enabled by default for installations with vanilla configuration (i.e., when compiled from source or installed via package managers).

Find the location of <conf_dir>

To get the location of `<conf_dir>`, use **`eval $${conf_dir}`** in `fs_cli` or `fs_cli -x 'eval $${conf_dir}'` on your terminal.

See Command-Line Interface (`fs_cli`) and the available configuration options.

TODO There is no real link for configuration options - the closest would be Global Variables (which is just a list) and Default Configuration (which only implicitly defines `conf_dir`). So update them.

The default `mod_event_socket` configuration binds to `::` (i.e., to listen to connections from **any** host), which will work on IPv4 or IPv6.

`::` means that `mod_event_socket` will listen to connections from **any** host (see vanilla `<conf_dir>/autoload_configs/event_socket.conf.xml` configuration file in the SignalWire GitHub repository). **There are obvious security risks**, so one would want to change this (e.g., to localhost only, `::1`), and perhaps also limit access via a firewall and/or ACL, as well as never using the default password.

```
<configuration name="event_socket.conf" description="Socket Client">
  <settings>
    <!-- Allow socket connections from any host -->
    <!-- :: is the IPv6 version of 0.0.0.0 in IPv4 -->
    <param name="listen-ip" value="::"/>
    <param name="listen-port" value="8021"/>
    <param name="password" value="ClueCon"/>
  </settings>
</configuration>
```

To change the default behaviour, use the `listen-ip` parameter. For example, to allow connections from localhost only:

```
<configuration name="event_socket.conf" description="Socket Client">
  <settings>
    <param name="nat-map" value="false"/>
    <!-- ::1 is the IPv6 version of 127.0.0.0/8 in IPv4 -->
    <param name="listen-ip" value="::1"/>
    <param name="listen-port" value="8021"/>
    <param name="password" value="ClueCon"/>
    <!--<param name="apply-inbound-acl" value="loopback.auto"/>-->
    <!--<param name="stop-on-bind-error" value="true"/>-->
  </settings>
</configuration>
```

IPv6 support

You can listen on IPv6 addresses (with the `listen-ip` parameter) since revision 8c794ac.

For example, the above example using `::` (IPv6 equivalent of 0.0.0.0) will enable listening on both IPv4 and IPv6 addresses on dual stack hosts.

1.1 ACL

Using Access Control List (ACL)s is recommended for enhanced security if you allow ESL connections from another machine. One can either enable named Access Control List (ACL)s (defined in `<conf_dir>/autoload_configs/acl.conf.xml`) or allow IP ranges directly in `event_socket.conf.xml`:

Syntax

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

Examples

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

Multiple `apply-inbound-acl` params will **not** work.

As of 1.6 you must supply an ACL. In order to allow all IPs you can use **`any_v4.auto`** in `event_socket.conf.xml`.

2. Modes of operation

2.1 Inbound mode

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

```
    *********************************************
    *                     |                     *
    *  FreeSWITCH™        |  mod_event_socket   *
    *                     |  127.0.0.1:8021     *
    *                     |                     *
    *********************************************
              /\                   /\
              /                     \
       ******************       ******************
       * Client A       *       * Client B       *
       * 127.0.0.1:9988 *       * 127.0.0.1:9999 *
       ******************       ******************
```

In inbound mode, your application (Client A: Python, Client B: 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.

If you would like to handle incoming calls using inbound mode, you should add the `uuid_park` command (see mod_commands) to your dialplan. Otherwise the dialplan might complete executing before your client can send commands to the event socket.

2.2 Outbound mode

Outbound mode means you make a daemon (with whatever language), and then have FreeSWITCH 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.

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.

3. Command Documentation

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

3.1 api

Send a FreeSWITCH API command, **blocking mode**. That is, the FreeSWITCH instance won't accept any new commands until the `api` command finished execution.

Usage

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

Examples

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

3.2 bgapi

Send a FreeSWITCH 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.

Usage

```
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
```

`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
```

3.3 linger

Tells FreeSWITCH not to close the socket connection 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.

Usage

```
linger
```

3.4 nolinger

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

Usage

```
nolinger
```

3.5 event

The `event` command is used to subscribe on events from FreeSWITCH (plain text, XML, or JSON output format)

Usage

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

You may specify any number events on the same line that should be separated with spaces.

See Event List page for a comprehensive list of events.

Examples

```
event plain ALL
event plain CHANNEL_CREATE CHANNEL_DESTROY CUSTOM conference::maintenance sofia::register sofia::expire
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`,

```
event plain DTMF
```

but you want to register for `CHANNEL_ANSWER` also, then it is enough to do

```
event plain CHANNEL_ANSWER
```

and you will continue to receive `DTMF` along with `CHANNEL_ANSWER`.

3.5.1 `event plain` format

Events consist of

- a **header section**, and
- an optional **event body**.

3.5.1.1 Header section

Headers are key/value pairs separated by a colon, and headers are separated by 1 line feed.

Some header values may contain multiple line breaks, but because FreeSWITCH URL-encodes all of them, these multi-line header values will still appear as 1 line.

Example of multi-line header values

```
# header BEFORE URL-encoding
variable_switch_r_sdp: v=0
o=UAC 6407 6867 IN IP4 192.168.27.72
s=SIP Media Capabilities
c=IN IP4 61.231.8.102
t=0 0
m=audio 12916 RTP/AVP 0 18 101
a=rtpmap:0 PCMU/8000
a=rtpmap:18 G729/8000
a=fmtp:18 annexb=no
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=maxptime:20

# header AFTER URL-encoding
variable_switch_r_sdp: v%3D0%0D%0Ao%3DUAC%206407%206867%20IN%20IP4%20192.168.27.72%0D%0As%3DSIP%20Media%
20Capabilities%0D%0Ac%3DIN%20IP4%2061.231.8.102%0D%0At%3D0%200%0D%0Am%3Daudio%2012916%20RTP/AVP%200%
2018%20101%0D%0Aa%3Drtpmap%3A0%20PCMU/8000%0D%0Aa%3Drtpmap%3A18%20G729/8000%0D%0Aa%3Dfmtp%3A18%20annexb%
3Dno%0D%0Aa%3Drtpmap%3A101%20telephone-event/8000%0D%0Aa%3Dfmtp%3A101%200-15%0D%0Aa%3Dmaxptime%3A20%0D%
0A
```

3.5.1.2 Event body

Events may have a body, carrying additional content generated with the event. It is usually not in the key/value form of headers, and may contain its own native formatting.

The presence of `Content-Length` in the **header section** is an indicator that there is an **event body**. The value of the `Content-Length` header is the size of the **body** in bytes.

(e.g., DETECTED_SPEECH event example from Event List page

```
Speech-Type: detected-speech
Event-Name: DETECTED_SPEECH
Core-UUID: aac0f73e-b822-e54c-a02a-06a839ca3e5a
FreeSWITCH-Hostname: AMONROY
FreeSWITCH-IPv4: 192.168.1.220
FreeSWITCH-IPv6: ::1
Event-Date-Local: 2009-01-26 16:07:24
Event-Date-GMT: Mon, 26 Jan 2009 22:07:24 GMT
Event-Date-Timestamp: 1233007644906250
Event-Calling-File: switch_ivr_async.c
Event-Calling-Function: speech_thread
Event-Calling-Line-Number: 1758
Content-Length: 435

<result grammar="<request1@form-level.store>#nombres">
        <interpretation grammar="<request1@form-level.store>#nombres" confidence="0.494643">
                <instance confidence="0.494643">
                        arturo monroy
                </instance>
                <input mode="speech" confidence="0.494643">
                        <input confidence="0.313102">
                                arturo
                        </input>
                        <input confidence="0.618854">
                                monroy
                        </input>
                </input>
        </interpretation>
</result>
```

Event structure

```
Content-Length: <size>\n                                          | Headers in the TCP/IP
Content-Type: text/event-plain\n                                  | packet's payload.
\n
event-hdr1: a\n       <-- size starts here    | FreeSWITCH       | Event headers and body
event-hdr2: b\n                               | event headers    | in the the body of the
...                                           |                  | TCP/IP packet's body.
event-hdrN: x\n                               *----------------  |
\n                    <-- size ends here                         |
body line 1               (if no body)        *----------------  |
...                                           | FreeSWITCH       |
body line N           <-- or here if there's  | event body       |
                          a body              |                  |

--------------------------------------------------------------------
\n is a line feed in the form of CRLF.
```

Example event: BACKGROUND_JOB

```
Content-Length: 625
Content-Type: text/event-plain

Job-UUID: 7f4db78a-17d7-11dd-b7a0-db4edd065621
Job-Command: originate
Job-Command-Arg: sofia/default/1005%20'%26park'
Event-Name: BACKGROUND_JOB
Core-UUID: 42bdf272-16e6-11dd-b7a0-db4edd065621
FreeSWITCH-Hostname: ser
FreeSWITCH-IPv4: 192.168.1.104
FreeSWITCH-IPv6: 127.0.0.1
Event-Date-Local: 2008-05-02%2007%3A37%3A03
Event-Date-GMT: Thu,%2001%20May%202008%2023%3A37%3A03%20GMT
Event-Date-timestamp: 1209685023894968
Event-Calling-File: mod_event_socket.c
Event-Calling-Function: api_exec
Event-Calling-Line-Number: 609
Content-Length: 41

+OK 7f4de4bc-17d7-11dd-b7a0-db4edd065621
```

3.5.2 event `plain` parsing instructions

1. **Look for 2 line feeds** or end-of-line sequences(EOL)
   TODO The original text contains LF, the FreeSWITCH book says CRLF. Figure out which is correct.

2. **Get the event data**
   Read exactly as many bytes from the socket as specified in the `Content-Length` header

   Note that since this is TCP, this may take more than one read so if you are supposed to read 200 bytes and the next read only returns 50, you must continue to read another 150, and so on until you have read 200 bytes or the socket has an error.

   Once you have read all the bytes in the `Content-Length` header, the next packet will start on the subsequent byte.

3. **Parse event data**
   Is the `Content-Length` header among the event headers? (Implying that there is an event body.)

   a. **No** All done.
   b. **Yes**
      i. Evaluate `Content-Length` value
      ii. Read that many bytes

      `Content-Length` is the length of the event (in bytes) beginning **AFTER** the double LF line ("\n\n") of the event header.

3.5.3 Special Cases

3.5.3.1 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.

3.5.3.2 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 gtalk channel, ASR events and others.

3.6 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
```

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

3.7 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.

3.8 sendevent

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

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

<body>
```

Some phones require authentication for NOTIFY requests. FreeSWITCH can respond to a digest challenge if reverse authentication credentials are supplied for the user. See XML User Directory.

Uncategorized content

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

`sendevent CHANNEL_HANGUP`

Something that was undocumented but is supported; SIP INFO messages can be sent to every IP you need.

```
sendevent SEND_INFO
profile: external
content-type: text/plain
to-uri: sip:1@2.3.4.5
from-uri: sip:1@1.2.3.4
content-length: 15

test
```

3.8.1 Examples

3.8.1.1 Switch phone MWI led (tested on yealink)

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)
```

No Message:

MWI No Message

```
sendevent  message_waiting
MWI-Messages-Waiting: no
MWI-Message-Account: sip:user1@192.168.1.14
```

Some Message:

MWI With Message

```
sendevent  message_waiting
MWI-Messages-Waiting: yes
MWI-Message-Account: sip:user1@192.168.1.14
MWI-Voice-Message: 0/1 (0/0)
```

3.8.1.2 Make Snom phones reread their settings from the settings server

Snom NOTIFY

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

3.8.1.3 `sendevent` examples with a message body

The length of the body is specified by the `Content-Length` header.

```
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
```

or

Send Message

```
sendevent SEND_MESSAGE
profile: internal
content-length: 2
content-type: text/plain
user: 1005
host: 99.157.44.194

OK
```

Results in a packet like this:

Send Message result

```
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, INFO,
PUBLISH
Supported: 100rel, timer, precondition, path, replaces
Content-Type: text/plain
Content-Length: 2

OK
```

3.8.1.4 SIP Proxy Example

To send through a proxy, use the event headers: contact-uri, to-uri, and from-uri
In this example, the SIP proxy is 192.168.207.156:5060 and the UA can be reached at 1002@192.168.0.99:11710

```
sendevent NOTIFY
profile: internal
content-type: application/simple-message-summary
event-string: check-sync
user: 1002
host: 3.local
to-uri: 1002@3.local
from-uri: 1002@3.local
contact-uri: sip:1002@192.168.0.99:11710;fs_path=sip:192.168.207.156:5060
```

Another example with a notify:

NOTIFY Example

```
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
```

Results in a packet like this:

NOTIFY Result

```
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, INFO,
PUBLISH
Supported: 100rel, timer, precondition, path, replaces
Event: check-sync
Allow-Events: talk, presence, dialog, call-info, sla, include-session-description, presence.winfo, message-
summary
Subscription-State: terminated;timeout
Content-Type: application/simple-message-summary
Content-Length: 2

OK
```

3.8.1.5 Sipura/Linksys/Cisco phone or ATA to resync config with a specified profile URL

Cisco Resync

```
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
to-uri: sip:1000@10.20.30.40
from-uri: sip:1000@10.20.30.40
```

Results in a packet like this:

Cisco Resync result

```
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, SUBSCRIBE
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, presence.
winfo, message-summary, refer
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 then the unit will use its stored profile URI

3.8.1.6 Example usage for CSTA event:

CSTA Event

```
sendevent SWITCH_EVENT_PHONE_FEATURE
profile: internal
user: ex1004
host: 3.local
device: ex1004
Feature-Event: DoNotDisturbEvent
doNotDisturbOn: on
```

3.8.1.7 Display a text message on a Snom 370 or Snom 820

The message must be of type "text/plain".

3.9 sendmsg

Usage

```
sendmsg <UUID>
<headers>

<body>
```

`sendmsg` is used to control the behavior of FreeSWITCH. `UUID` is mandatory, and it refers to a specific call (i.e., a channel or call leg or session; see Call Legs and Creating a New Endpoint: Lifecycle of a Session).

TODO What does this line mean? "*originate a call directly to park by making an ext the ext part of the originate command &park()*"

All `sendmsg` commands must be followed by 2 returns.

Since messaging format is similar to RFC 2822, if you are using any libraries that follow the line wrapping recommendation of RFC 2822 then make sure that you disable line wrapping as FreeSWITCH will ignore wrapped lines. See "FreeSWITCH EventSocket header length" post for more.

Example without a body (note the extra line feed at the end!)

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

```

3.9.1 Commands

3.9.1.1 execute

`execute` is used to invoke dialplan applications, mirroring the function of the `ESLconnection` object's `execute` method (in the Event Socket Library).

Check the XML Dialplan and `mod_dptools` pages for available applications.

Force synchronous operations in async mode

If you are using `async` outbound mode, you need to pay attention to potential race conditions, as the commands you send may not execute in sequential order.

You may force the command to wait by setting `event-lock` until the critical or long-running command finishes:

```
sendmsg
call-command: set
execute-app-name: foo=bar\n\n
event-lock:true
```

TODO Is `event-lock` common to all commands? Examples here in mod_event_socket and most in Event Socket Outbound are using `sendmsg` with `call-command: execute`, and `setEventLock` in Event Socket Library also only mentions `execute`, yet Event Socket Outbound also shows the above snippet.

TODO Is `call-command: set` still an available command? Taken from Event Socket Outbound, but haven't seen documented anywhere else, and it is also not listed in the source. Was the intention something like this?

```
sendmsg
call-command: execute
execute-app-name: set
execute-app-arg: foo=bar\n\n
event-lock: true
```

The format should be:

Execute format

```
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>
```

When an application is executed via `sendmsg`, `CHANNEL_EXECUTE` and `CHANNEL_EXECUTE_COMPLETE` events are going to be generated. If you would like to correlate these two events then add an `Event-UUID` header with your custom UUID. In the corresponding events, the UUID will be in the `Application-UUID` header. If you do not specify an `Event-UUID`, Freeswitch will automatically generate a UUID for the `Application-UUID`.

Example:

`Event-UUID: 22075ce5-b67b-4f04-a6dd-1726ec14c8bf`

3.9.1.2 hangup

Hang up the call.

Format

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

Additional information

- Hangup Causes

3.9.1.3 `unicast`

`unicast` is used to hook up `mod_spandsp` for faxing over a socket.

Brian's note

> *That is a nice way for a script or app that uses the socket interface to get at the media. It's good because then spandsp isn't living inside of FreeSWITCH and it can run on a box sitting next to it. It scales better.*

unicast example

```
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
```

3.9.1.4 `nomedia`

No description.

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

3.9.1.5 xferext

⚠ TODO Document this command.

From Stéphane Alnet's comment:

> *The description is missing `xferext`, which adds `application` to an extension then transfer the channel to that extension (whatever that means! 🤐).*

`switch_ivr.c:503` lists the available commands, and that C file also contains the logic.

src/switch_ivr.c

```
        unsigned long CMD_EXECUTE = switch_hashfunc_default("execute", &hlen);
        unsigned long CMD_HANGUP = switch_hashfunc_default("hangup", &hlen);
        unsigned long CMD_NOMEDIA = switch_hashfunc_default("nomedia", &hlen);
        unsigned long CMD_UNICAST = switch_hashfunc_default("unicast", &hlen);
        unsigned long CMD_XFEREXT = switch_hashfunc_default("xferext", &hlen);
```

3.10 exit

```
exit
```

Close the socket connection.

3.11 auth

```
auth <password>
```

3.12 log

```
log <level>
```

Enable log output, and change the log level.

3.13 nolog

```
nolog
```

Disable log output previously enabled by the log command

3.14 nixevent

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

Suppress the specified type of event. Useful when you want to allow 'event all' followed by 'nixevent <some_event>' to see all but 1 type of event.

3.15 noevents

```
noevents
```

Disable all events that were previously enabled with event.

See Also

- Event Socket Library
- Event Socket Outbound
- Debugging Event Socket Message [old wiki]
- Making Event Socket behave like the console
- Event List
- Email2callback - Email to callback application with Python and freepy.