mod_sms

About

This page is working on progress ....

mod_sms provide a way to route messages in freeswitch, potentially allowing one to build a powerful chatting system like in XMPP using using SIP SIMPLE on SIP clients.

Just like XML Dialplan has dialplan, mod_sms has chatplan.

mod_sms bind on GLOBAL message event system, so it catches all MESSAGE events and then route them to the chatplan. If no chatplan entry matches, it works as default - e.g. two clients sending message in a point-to-point manner.

Compile and Load

In the source tree

```
make mod_sms-install
```

In fs_cli

```
load mod_sms
```

Config

mod_sms is a new module (in late Sept 2011), so users who upgrade from an older FreeSWITCH install need to copy freeswitch.xml from source or add the following section to the existing conf/freeswitch.xml.

```
<section name="chatplan" description="Regex/XML Chatplan">
 <X-PRE-PROCESS cmd="include" data="chatplan/*.xml"/>
</section>
```

You'll also need to add mod_sms to the loadable modules in conf/autoload_configs/modules.conf.xml

```
<load module="mod_sms"/>
```

And then create the conf/chatplan dir. Create a default.xml like this:

```
<?xml version="1.0" encoding="utf-8"?>
<include>
 <context name="default">
  <extension name="demo">
   <condition field="to" expression=".*$"/>
   <action application="reply" data="Hello, you said: ${_body}"/>
  </condition>
 </extension>
</context>
</include>
```

Basic Usage With Default Config

I used Eyebeam for test, I believe Xlite should work. The following demo shows a message sent from 1004 to 1019.
Create new contact in Eyebeam and then right click the contact and click instant message then an instant message window will appear.

Type hello and you should get a reply immediately "Hello, you said hello"

The log will like this:

<table>
<thead>
<tr>
<th>Time</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-09-29</td>
<td>Processing text message 1004-&gt;1019 in context default</td>
</tr>
<tr>
<td>2011-09-29</td>
<td>looking for bindings</td>
</tr>
<tr>
<td>2011-09-29</td>
<td>no binding for chatplan</td>
</tr>
<tr>
<td>Chatplan: 1019</td>
<td>continue=false</td>
</tr>
<tr>
<td>Chatplan: 1019</td>
<td>Regex (PASS) demo to(1019@192.168.0.81) =~ /^(.*)$/ break=on-false</td>
</tr>
<tr>
<td>Chatplan: 1019</td>
<td>Action reply(Hello, you said: ${_body})</td>
</tr>
</tbody>
</table>

From the log you could find out that the message was caught by mod_sms and then hit the chatplan. The default chatplan just simply sent back a reply.

**Chatplan Tools**

Just like dialplan tools there's a set of chatplan tools

**reply**

Reply to a message

```xml
<action application="reply" data="unsolicited reply"/>
```

**fire**

fire an event to the FS event system, here's a sample event:

```xml
<action application="fire" data=""/>
```

```
Event-Name: MESSAGE
Core-UUID: c0305628-761e-4a01-be5e-c39959f55b47
FreeSWITCH-Hostname: seven-macpro.local
FreeSWITCH-Switchname: seven-macpro.local
FreeSWITCH-IPv4: 192.168.0.81
FreeSWITCH-IPv6: ::1
Event-Date-Local: 2011-09-29 11:55:21
Event-Date-GMT: Thu, 29 Sep 2011 03:55:21 GMT
Event-Date-Timestamp: 1317268521199214
Event-Calling-File: sofia_presence.c
Event-Calling-Function: sofia_presence_handle_sip_i_message
Event-Calling-Line-Number: 2995
login: sip:mod_sofia@192.168.0.81:5060
proto: sip
from: 1004@192.168.0.81
from_user: 1004
from_host: 192.168.0.81
to_user: 1019
to_host: 192.168.0.81
from_sip_ip: 192.168.0.81
from_sip_port: 13206
to: 1019@192.168.0.81
subject: SIMPLE MESSAGE
type: text/html
from_full: "Seven3" <sip:1004@192.168.0.81>;tag=1c777f4c
sip_profile: internal
dest_proto: sip
DP_MATCH: 1019@192.168.0.81
Content-Length: 49
Content-Length: 49
```
send
Send the message as is.

```
<action application="send"/>
```

set
Set vars.

```
<action application="set" data="var=val"/>
```

stop
Stop execution.

```
<action application="stop" data=""/>
```

Programming
You can also directly call the various programming language modules from the chatplan:

```
<action application="lua" data="handle_chat.lua"/>
<action application="python" data="handle_chat.py"/>
```

To prevent the default behavior of sending the message directly even after a chatplan action has matched, set `final_delivery` to `true`.

```
<action application="set" data="final_delivery=true"/>
```

Interface
These scripts are run directly inside the chatplan. They have to define a function called "chat", which is then run. Here's an example from python:

```
def chat(message, args):
    sys.stderr.write(str(message.serialize()))
```

This interface defines two objects: Message and Args.

**Message**
Like the `session` object in the event system, there's a `message` object in chatplan. The primary functions are:

- `getHeader` — return one of the message headers
- `serialize` — convert the entire message into a string

Here's a sample of handle_chat.lua

```
freeswitch.consoleLog("info", message:getHeader("from"));
freeswitch.consoleLog("info", message:getHeader("to"));
freeswitch.consoleLog("info", message:serialize());
```

`message` is a standard event so it's the same API as always with an extra method `chat_execute` to run the apps.
message:chat_execute("reply", "you said: " .. message:getBody());

Args

Args is a string passed to the method containing whatever arguments were passed from the chatplan.

Global Vars

These scripts also have direct access to the variables present in the chatplan. This is done just as done elsewhere: GetGlobalVariable.

Sending a Message from a script

Though the send and reply functions exist, you may want more direct control over the output of the message. Instead, you can directly create and fire off a message event. The following is an example from Lua:

```lua
freeswitch.consoleLog("info", "chat console\n")
local event = freeswitch.Event("CUSTOM", "SMS::SEND_MESSAGE");
event:addHeader("proto", "sip");
event:addHeader("dest_proto", "sip");
event:addHeader("from", "1004@192.168.0.81");
event:addHeader("from_full", "sip:1004@192.168.0.81");
event:addHeader("to", "sip:1019@192.168.0.81");
event:addHeader("subject", "sip:1019@192.168.0.81");
event:addHeader("type", "text/html");
event:addHeader("hint", "the hint");
event:addHeader("replying", "true");
event:addHeader("sip_profile", "external");
event:addBody("Hello from Seven Du! Have fun!");
-- freeswitch.consoleLog("info", event:serialize());
event:fire();
```

Sending a Message via ESL

You can also send a message via ESL. mod_sms must be loaded or this will not work.

Here is an example perl script utilizing ESL to send a message.

```perl
require ESL;
my $con = new ESL::ESLconnection("localhost", "8021", "ClueCon");

my $e = new ESL::ESLevent("custom", "SMS::SEND_MESSAGE");
$e->addHeader("to", "1019@www.freeswitch.org");
$e->addHeader("from", "testing@foobar.com");
$e->addHeader("sip_profile", "internal");
$e->addHeader("dest_proto", "sip");
$e->addBody(shift);
$con->sendEvent($e);
```

And the same script in Python.

```python
require ESL;
my $con = new ESL::ESLconnection("localhost", "8021", "ClueCon");

my $e = new ESL::ESLevent("custom", "SMS::SEND_MESSAGE");
$e->addHeader("to", "1019@www.freeswitch.org");
$e->addHeader("from", "testing@foobar.com");
$e->addHeader("sip_profile", "internal");
$e->addHeader("dest_proto", "sip");
$e->addBody(shift);
$con->sendEvent($e);
```
from ESL import *

con = ESLconnection("127.0.0.1", "8021", "ClueCon")

event = ESLevent("CUSTOM", "SMS::SEND_MESSAGE")
event.addHeader("to", "1019@www.freeswitch.org")
event.addHeader("from", "testing@foobar.com")
event.addHeader("dest_proto", "sip")
event.addBody("message contents")

con.sendEvent(event)