2024

JVoIP -Java VoIP SDK

A full featured, flexible SIP client in a single JAR file

The Mizu Java VoIP SDK (JVoIP) is a lightweight standards based VoIP phone that can be used as a library or as a standalone application/applet. Based on the industry standard SIP, RTP and related protocols, it is compatible with all common VoIP devices, servers and softphones, providing easy integration capabilities with any application.

-
Mizutech
6/20/2024

Contents

About

The Mizu Java VoIP SDK (JVoIP) is a SIP client implemented as a platform independent java library. Since it is based on the open standard Session Initiation Protocol, it can inter-operate with any other SIP-based device (servers and clients).

The VoIP SDK can be used in many ways:

- as a library added to your project (use the API to create a SIP app or add VoIP capabilities for your app; this is the main use-case)
- as a command line Java VoIP client (with stdin input / stdout output)
- as a standalone desktop application (it has a built-in minimalistic GUI to ease such kind of usage)
- as an applet embedded to a web page (applets are already deprecated in modern browsers but still supported by JVoIP)

With the Java VoIP SDK you have an easy to use full featured SIP/media stack in a single jar file, easy to integrate or embedded in your desktop, server or web application. For example it can be integrated with callcenter software or embedded in VoIP devices such as PBX or gateways so users will have a fully functional VoIP softphone without the need to download any other third-party software. You can also use it to add VoIP call capabilities into any software not directly related to VoIP (such as games or CRM's) or to perform any kind of VoIP automation (auto dialer, auto answer machine, etc).

See all related resources here.

Requirements

JVoIP hardware/software requirements:

- Any OS with Java SE support (Linux, Windows, MAC, others)
- JVM: Works with most JVM's including OpenJDK (or from here), Oracle, Microsoft, Adoptium, Azul and others.
- Minimum Java version: J2SE 5.0+ (Java 1.5+ which means support for all java versions since 2004)
- Maximum Java version: any (22+). No limitations, the library doesn't use special modules or API's but will take advantage of the latest Java features only via a class loaded with proper failback. Java 22 is also fully supported (the current/latest Java when this documentation was last updated) and will remain compatible with all further Java JVM/JRE/JDK releases such as the not yet released Java SE 25.
- Development: any java compiler, any OS and any IDE can be used (the library is a single jar file which can be easily added to your project regardless of your environment)
- Programming language (when used as a library): any JVM based language such as Java, Clojure, Kotlin, Scala, Groovy, Python, Jython and others
- IDE (when used as a library): any IDE or command line can be used such as IntelliJ, Eclipse, NetBeans, JBuilder, JDeveloper, JCreator, BlueJ, Kite and others
- Audio device: headset or microphone/speaker for audio (will work also without audio device with streaming or voice recording)
- Video device: optional if you wish to use video calls. Any webcam device which is recognized by your OS
- CPU: any architecture with Java SE support such as x86, x64, arm (armv7k, arm64, etc), sparc, etc. with a performance of minimum 350 MHz Pentium 3 or higher (runs well also on ancient devices or with embedded devices such as Raspberry Pi)
- RAM: minimum 10 MB to run JVoIP (above the JVM basic requirements)
- Disk space: 3 MB (additional data such as call recording or detailed logs might take more space)
- A SIP account (At any VoIP service provider or your own IP-PBX/Softswitch/SIP sever. Can be also used without registration for peer to peer SIP calls)

Note: If you are looking for a SIP stack for your Android project, then you should use the AJVoIP SDK instead of this library (AJVoIP has the same API as this JVoIP SDK, but targets Android instead of Java SE)

Usage

The SDK can be used in many ways: as a standalone application, embedded into your project or integrated with your website.
The settings can be specified from: API, command line, config file, URL or sent via SIP signaling.
It can be used as a standalone app or integrated into other app with or without using the API.
The API can be accessed from: Java and any other JVM language, JavaScript, UDP, TCP or HTTP (clear text, URL, JSON, XML).
The API can be directly used from Java or other JVM based application such as Kotlin, or you can use the API via UDP/TCP or HTTP wrapper from any development environment and programming language such as C, C++, C# or Delphi.

Download: Java SIP SDK (this is the demo version)

Note: Optionally you might also copy the mediaench files to your app folder (near your java or jar files). This contains some platform dependent native binaries to optimize audio processing and the sip client will select the correct one to be used automatically if found.

Using from the console/command line

This is a simplest way to test as a simple SIP client and it is often used for various automation tasks:
Example:
java –jar JVoIP.jar serveraddress=VOIP_SERVER_IP_OR_DOMAIN username=USERNAME password=PASSWORD callto=DESTINATION autocall=false loglevel=1
Here is a working example: JVoIP.jar serveraddress=voip.mizu-voip.com username=jvoiptest password=jvoiptestpwd callto=testivr3 autocall=false loglevel=1
Replace the values in uppercase accordingly and make sure to have Java installed.

To disable the GUI (use as a console application) set the iscommandline JVoIP parameter to true. Otherwise you can launch the GUI also from command line with the API_StartGUI function (use API_StartGUI() instead of API_Start() if you need this functionality).
By default all parameters are set to its optimal/most common values, however if you wish to further customize or you need any specific functionality then you might use any other command line options as documented in the parameters chapter.

Note:
- The demo version doesn't work in headless mode like Linux with no X-Server/X-Window installed. Contact us if you need a headless version!
- Some systems (Windows) might accept the command line also without the java –jar prefix, other systems (Linux) might require the java –jar JVoIP.jar [parameters] format.

Using as a standalone application

This means starting the app with its built-in user interface.
Just double click on JVoIP.jar to start (or launch it from command line as java –jar JVoIP.jar)

It is also possible to create a configuration file near the JVoIP.jar with the file name: wpcfg.ini. The content should be like this:
```
serveraddress=VOIP_SERVER_IP_OR_DOMAIN
username=USERNAME
password=PASSWORD
callto=DESTINATION
loglevel=1
…other config options if needed
```
Replace the values in uppercase accordingly and make sure to have Java installed.

JVoIP includes this minimalistic softphone user interface for your convenience. It can be modified by the appearance parameters.
However, you can easily create your own user interface/design and use JVoIP as a SIP library in your app, as discussed below.

Using as a library

You can use JVoIP as an SDK if you are a developer to add VoIP functionality in any Java (or any other JVM based) application by including this SIP library into your project.
This is the most powerful use-case to fully exploit JVoIP capabilities.
You just need to add JVoIP to your project and call its public API_XXX functions. (See the "API" chapter below for more details").

In short, the interaction with the library is done by calling it's API functions, handling the function return values (success-true/failed-false/other answers) and handling the notifications received from the library (for example the STATUS messages about the SIP stack state machine).

Example use-cases:
- Implement your custom Java SIP client
- Implement a full featured Java Softphone
- Add VoIP capabilities to any Java application (or any JVM based apps such as Clojure, Scala, Kotlin, Groovy, JRuby or Jython)
- Add SIP call capabilities to any application running over a Java virtual machine
- Extend your SIP server with an intelligent extension (for example for call recording, IVR, RTP streaming, etc)

Steps:
1. Add JVoIP.jar lib to your project and add to your imports (import webphone.*;)
2. Extend the SIPNotificationListener class (class MyNotificationListener extends SIPNotificationListener) and override (some of) its members later if you wish to receive the Notifications
3. Instantiate a webphone object (webphone wobj = new webphone();)
4. Subscribe to notifications (webphoneobj.API_SetNotificationListener(new MyNotificationListener());)
5. Call the API_SetParameter() to pass any settings (parameters)
6. Call the API_Start() to start the SIP stack
7. Call any other functions (such as API_Call, API_SendChat and others as described in the API)

Note: all API calls are thread safe and will not throw exceptions (on exception they will send an "ERROR" notifications and/or return false/-1/0 depending on the context). To handle errors, check the functions return values, check the notifications, query the SIP stack state with various API calls and inspect the logs.

Example code (exception and error handling removed for simplicity):

```
package yourpackagename;
import webphone.*;  //add JVoIP.jar to your project for this. both the package and the main class are named "webphone"

//create webphone class object instance:
webphone wobj = new webphone();

//subscribe to notifications:
webphoneobj.API_SetNotificationListener(new MyNotificationListener());
```

```
//set parameters (replace uppercase words):
wobj.API_SetParameter("serveraddress", "VOIP_SERVER_IP_OR_DOMAIN");
wobj.API_SetParameter("username", "SIP_USERNAME");
wobj.API_SetParameter("password", "SIP_PASSWORD");
wobj.API_SetParameter("loglevel", "5");  //you might set to 1 for production
//you might set other parameters here. most parameters can be also set or changed later at runtime
//start the sipstack:
wobj.API_Start();
//register to your SIP server (optional):
//wobj.API_Register();

//make a call to a user/extension/phone number/SIP URI (note: 1-2 seconds might be needed between API_Start/API_Register and API_Call for the sipstack to initialize)
wobj.API_Call(-1, "DESTINATION");
//the following lines should be used from another function
//during the call you might call call divert functions by your app logic or on user interaction, for example API_Hold()

//call hangup to end the call (possibly triggered by a "Hangup" button pressed by the user):
wobj.API_Hangup(-1);
//stop JVoIP when you don't need it anymore
wobj.API_Stop();

//handle the notifications:
class MyNotificationListener extends SIPNotificationListener
{
  //see the Notifications chapter and the javadoc for the details.

  public void onAll(SIPNotification e) {
    System.out.println("Notification received: " + e.toString());
  }

  //override any other members here after your needs.
}
```

You can download a working example from here.

Contact us for the headless version if you wish to use it from an operating system without GUI system installed (such as Linux without X-Server/X-Window installed or used with a headless JRE/JDK).

Socket/HTTP API

The Socket interface is useful if you wish to use JVoIP as a library from non-JVM environment such as .NET, C++, Delphi, PHP, Python or any others.

While from Java (or other JVM based language) you can call the API functions directly, in case if you are using other development environment then you must call the JVoIP API via a socket. In this case you will have to convert your function calls to a simple string and send it over a socket. You will receive the answers for your API requests and notifications as strings over the same socket. The send/receive string formats are discussed below.

In short:
1. Launch JVoIP in background from your app
2. Send API requests via UDP packets or TCP socket as described here
3. Receive the notifications as stings over UDP or TCP

All the functions and notifications are the exact same like you would use JVoIP as an embedded library, the only difference is that the library will run in a separate process, you connect to it via UDP, TCP or HTTP and you have to convert/parse everything (API requests, API answers and notifications) as strings.

Configuration:
First of all, you will have to launch JVoIP as a separate process (for example ShellExecute on Windows, but for first you can just launch it manually from command line), then send commands to the JVoIP listening port by:

- UDP (configurable by the wpapiudplistenport parameter. 19422 by default)
- TCP (configurable by the wpapilistenport parameter. 0 by default which means disabled)
- HTTP (configurable by the same wpapilistenport parameter as it will auto detect raw TCP vs HTTP. 0 by default which means disabled)

Note:

- Only UDP is enabled by default. If you need TCP and/or HTTP set the wpapiudplistenport to 0 and the wpapilistenport to any value, for example 19422.
- By default it will listen on the localhost loopback address. You might modify it with the wpapilistenip parameter.
- Some very restrictive firewalls might block also localhost connections. If you have installed such a firewall on your PC, make sure to allow JVoIP/Java.
- If you start to use the API_SetNotificationListener, API_PollNotificationStrings or API_GetNotificationStrings function (using the API directly from Java) the JVoIP will stop sending out notifications via sockets

The following **formats** are accepted for requests and answers:

- clear text (so you can easily test it manually with telnet)
- HTTP (GET/POST/PUT/AJAX)
    - URL parameters (if used from a browser as an applet)
    - JSON
    - form post

- XML
- SOAP

Requests:
Example from browser (HTTP with URL parameters): http://127.0.0.1:19422/?function=API_Call&line=-1&peer=1234
Example from telnet (clear text TCP):

```
telnet 127.0.0.1 19422
function=API_Call&line=-1&peer=1234
```

Most of the documented functions are supported, except those that requires or returns buffers/streams or other objects (only string and number parameters/return values can be serialized over the socket API).
For the function parameters you can use the exact same name as in this documentation or you can use param1, param2 … paramN format.

Supported protocols
- raw UDP packets (separate UDP packets per requests).
- raw TCP (requests separated by \r\n or by EOFCOMMAND/BOFCOMMAND).
- HTTP (GET, POST, PUT requests)

Formatting
The function name and parameters can be formatted in the following ways:
- Clear text separated by comma (used with UDP or TCP).
  Example: function=API_TestEcho,echo=test
  It is also possible to use the following formats (but not recommended):
  - specify the parameters in order as param1, param2, …paramN: function=API_TestEcho,param1=test
  - or the shortest format by just separating the function name and parameters with comma: API_TestEcho,test
- Encapsulated between EOFCOMMAND/BOFCOMMAND and the parameters between EOFLINE/BOFLINE (used with UDP or TCP)
  This is recommended for more accurate processing.
  Example: BOFCOMMANDBOFLINEfunction=API_CallEOFLINEBOFLINEparam1=-1EOFLINEBOFLINEparam2=1234EOFLINEEOFCOMMAND
- URI parameters (commonly used with HTTP GET request, but supported also over UDP and TCP).
  Example: function=API_Call&line=-1&peer=1234
- With HTTP you can use any of the followings:
  - URI parameters (GET requests)
  - key/value pairs (with POST requests)
  - form post format
  - JSON
  - XML
  - SOAP

  Example (HTTP GET with URI parameters): http://127.0.0.1:19422/?function=API_Call&line=-1&peer=1234

From local applications we recommend using UDP with clear text. For more accurate processing you can also split the commands between EOFCOMMAND/BOFCOMMAND and the parameters between EOFLINE/BOFLINE.

Answers:
The answers are formatted in the same way as the request with clear text content. For example if you send a JSON request then you will receive a JSON answer, and so on.
Example answer content:
APIREQUEST:API_CallRPARAM1:1RPARAM2:1234APIRESULT:**RESULT**
The first part (until "APIRESULT:") might be useful if you are using the API asynchronously, so you will always know for which request did you received the answer. The important part is the RESULT string (after "APIRESULT:") and usually begins with OK or ERROR. For example: OK: initiated or ERROR: failed

Notifications:
If you are using simple UDP or TCP (not HTTP) then the notification strings are also sent automatically on the same socket. Otherwise you can use API_PollNotificationStrings to receive the notifications (polling).

From HTTP clients we recommend AJAX HTTP requests with URL parameters. Also you should set the polling parameter to 3 and use API_PollNotificationStrings to receive the notifications (since in case of HTTP there is no any live socket stream where the notification might be sent, it is recommended to constantly poll the JVoIP for these events using the API_PollNotificationStrings function from a timer in around every 200 milliseconds).

Summary by protocol:
- UDP:
  - You might change the wpapiudplistenport parameter if you wish. Default is 19422.
  - It is recommended to bind your UDP socket to 127.0.0.1:19421 (the socket in your own app, not for the JVoIP) and change your target port to the port from where you receive messages on UDP. This helps if the JVoIP was unable to bind to the specified port and is running on a different port as it will send messages to 19421 by default (and changes the target port to the source address once receives any API command).
  - Send commands as clear text in UDP packets to port 19422 (or to the port from where you received the last udp message from JVoIP)
  - You will receive the notifications events from the same port

- TCP:

- Set the wpapilistenport parameter to 19422
- Connect to port 19422 and send commands as strings
- You will receive the notifications events in the same TCP stream
- You can easily test with a telnet client by connecting to 127.0.0.1 19422 and sending “function=API_TestEcho&echo=test”

- HTTP:
  - Set the following JVoIP parameters:
    - wpapiudplistenport: 0
    - wpapilistenport: 19422
    - polling: 3
  - Send commands as HTTP GET requests to http://127.0.0.1:19422
  - You can use various formats such as XML or JSON (default is clear text)
  - You need to poll for notification events using API_PollNotificationStrings (from a timer in every 200 msec)
  - You can easily test with a browser: http://127.0.0.1:19422/?function=API_TestEcho&echo=teeeest

Using on a website

You can also use JVoIP as an Applet from browsers. This is deprecated now in modern browsers but there are special browsers with Applet support and other ways to benefit from this method (for example converting to WebAssembly).
See the details here.

Features

- Standard SIP client for audio/video calls (in/out), chat, conference and others
- **SIP** and RTP stack compatible with any VoIP server or client (Cisco, Asterisk, Twilio, gateways, ATA, softphones, IP Phones, X-Lite and many others)
- Protocols: SIP/SIPS, RTP/SRTP. IP layer: IPv4/IPv6
- Transport: automatic, UDP, TCP, TLS, TCP tunnel, SOCKS proxy traversal, HTTP proxy traversal, HTTP, VPN tunneling
- NAT/Firewall support: stable SIP and RTP ports ,keep-alive, UPnP, rport support, fast ICE/fast STUN protocols and auto configuration
- Encryption: **TLS/SRTP**, tunneling and peer to peer encrypted media
- RFC’s: 2543, 3261, 2976, 3892, 2778, 2779, 3428, 3265, 3515, 3311, 3911, 3581, 3842, 1889, 2327, 3550, 3960, 4028, 3824, 3966, 2663, 3022 and many other SIP related standards
- IMS/3GPP/VoLTE (basic compatibility and features such as USSD or 3GPP SMS)
- Supported methods: REGISTER, INVITE, reINVITE, ACK, PRACK, BYE, CANCEL, UPDATE, MESSAGE, INFO, OPTIONS, SUBSCRIBE, NOTIFY, REFER
- Audio codec: G.711 (PCMU, PCMA), **G.729**, GSM, iLBC, **OPUS**, SPEEX, G.722.1
- Video codec: H261, H263, H264, MPEG1, MPEG2, MPEG4, VP8, VP9, Theora
- HD Audio: Wideband, **ultra-wideband** and full-band codecs (opus, speex, G.722.1)
- Audio enhancements: Stereo output (will convert mono sources to stereo) , PLC (packet loss concealment), AEC (acoustic echo canceller), Noise suppression, Silence suppression, AGC (automatic gain control) and auto QoS
- **Conference** calls (built-in RTP mixer)
- **Voice recording** (local, FTP or HTTP upload in wav, mp3, gsm or ogg format), SIPREC, custom **audio/video streaming** (to external app or service)
- DTMF (INFO method in signaling or RFC2833)
- **IM/Chat** (RFC 3428), group chat, SMS (including 3GPP SMS support), **BLF** and **presence** capability
- Offline chat (late delivery if peer is offline)
- Redial, mute, call **hold**, MOH (music on hold), **forward** and **transfer** (unattended and attended, including supervised transfer with replaces)
- Call park and pickup, auto-answer, barge-in
- Balance display, call timer, inbound/outbound calls, Caller-ID display, Voicemail (MWI)
- Additional features: call parking, early media, 3PCC, ED-137, local ring-back, PRACK and 100rel, replaces
- High availability: auto server failover, transport failover, DNS SRV based failover, backup server configuration, auto-reconnect, network auto-recover
- A long list of other minor features (see the parameters, the API and the FAQ for details)
- Unlimited lines (multiple simultaneous calls), multiple accounts (register with multiple SIP accounts from the same instance)
- Ease of use: one single exported class file contains all the public API functions
- Flexibility (all parameters/behavior can be changed/controlled by parameters and/or the API)
- Cross-platform (any OS with Java SE support including Windows, MAC, Linux)
- Works with all JVM versions, including latest JVM/JRE/JDK (backward compatible until JVM v.1.5 / J2SE 5.0 which was released in 2004)
- Full backward/forward compatibility including the settings and the API (no settings or code changes are required when upgrading to new versions)

See the software homepage for a general presentation.

Licensing

The Mizu Java VoIP SDK (JVoIP) is sold with perpetual life-time unlimited client license (Advanced and Gold) or restricted number of licenses (Basic). You can use it with SIP server(s) which belongs to you or your company/organization according to your license.

You can find the licensing possibilities on the Java VoIP SDK page. After successful tests please ask for your final version at webphone@mizu-voip.com. Mizutech will deliver the JVoIP build within one workday after your payment.

Release versions don't have any of the demo limitations and can be fully customized with your branding with "mizu" and "mizutech" words and links removed. Your final build must be used only for you company needs (including your direct customers / sip endusers) and it must be incorporated into your software/service.
Most (99%) of the code was written by Mizutech developers from scratch, including both the SIP and media stack, except the optional third-party modules listed below, with the goal of creating a compact but full featured, robust and easy to use SIP library for Java. The SDK is free from any patents or legal obligation toward third-parties.

The following freely and legally distributable open-source optional third-party modules might be used in your copy:

- DNS: dnsjava under BSD license
- Opus: the open-source opus codec implementation transformed to Java under three-clause BSD license.
- Speex: the open-source speex codec implementation transformed to Java under revised BSD license.
- G.729: based on the open-source G.729 ITU-T implementation transformed to Java. Patent expired in 2017.
- G.722.1: royalty-free ITU-T standard G.722.1 (05/2005)
- GSM: open-source gsm codec from Tritonus under Apache license
- OGG/Vorbis: open-source implementation transformed to Java under BSD license.

The library doesn't have any external dependencies (the above libraries are also compiled into the .jar when applicable).

Title, ownership rights, and intellectual property rights in the Software shall remain with MizuTech and/or its suppliers.
The agreement and the license granted hereunder will terminate automatically if you fail to comply with the limitations described herein. Upon termination, you must destroy all copies of the Software. The software is provided "as is" without any warranty of any kind.

You may:

Use JVoIP on any number of computers (Advanced or Gold version)
Use JVoIP as an SDK embedded in your project
Give the access to functionalities provided by JVoIP for your customers or use within your company
Use JVoIP with VoIP servers for which you have license for (after the agreement with Mizutech). All the VoIP servers must be owned/controlled by you or your company. Otherwise, please contact our support to check the possibilities

You may not:

Resell JVoIP
Sell JVoIP as a standalone application (you are allowed to use it only coupled with your project which purpose should not be the same as JVoIP's)
Sell "JVoIP" services for third party VoIP providers and other companies
Use JVoIP with VoIP servers not communicated with Mizutech (except if you have a special agreement with the Gold license)
Reverse engineer, decompile or disassemble JVoIP
Modify the software in any way (except modifying the parameters and using it via the public API)

You shall not use the JVoIP software in any way that competes either directly or indirectly with Mizutech (such as creating a general purpose Java SIP library), including but not limited to creation of derivative works that compete either directly or indirectly with our JVoIP software.
You shall not sell or offer JVoIP or the derivative works as a software or as a service for other companies, organizations or individuals which competes either directly or indirectly with Mizutech JVoIP offering.

Oracle Java licensing:

JVoIP has little to do with Oracle licensing.
You can use it with any JVM, including old Oracle Java SE versions, the new free Oracle Java SE versions or any other compatible Java implementations such as OpenJDK, IceTea, AdoptOpenJDK or any JVM bundled with your OS.

Demo version:

We are providing a demo version which you can try and test before any payment. The demo version has all features enabled but with some restrictions to prevent commercial usage. The limitations are the followings:

- maximum 10 simultaneous JVoIP instances in the same time
- will expire after several months of usage (usually 2 or 3 months)
- maximum ~100 sec call duration restriction
- maximum 10 calls / session limitation (after 10 calls you will have to restart)
- will work only maximum 20 minutes and after that you have to restart the JVoIP library or your application
- verifications against the mizu license service
- no headless mode (contact us if you need a headless trial)

Note: for the first few calls there might be fewer limitations than described above.
On request we can also send a trial version (will expire after some time but doesn't have the above demo limitations).
To upgrade your demo/trial to an unlocked license, see the details here.

API

You can use JVoIP as an SDK directly embedding into your Java application. Additionally you can access the same API also from JavaScript or via UDP, TCP, HTTP from any application.

Functions

The SIP SDK exposes numerous API functions, however this doesn't mean that the usage is difficult. You can ignore most of these functions and use only those few relevant for your needs also outlined in the usage example. For example if you just need to make simple calls, then the following three functions will cover all your needs: API_Start, API_Call, API_Hangup.

*Most of the functions **return a boolean** value. True when the operation was completed or initiated successfully, otherwise false.*
*Some of the functions are **executed asynchronously** (API_Call, API_Register, etc). This means that it might return a true value immediately but it might fail later. For example for API_Call the return value means only that the call was initiated successfully. At this point we don't know if the call will be successful (connected) or not. You can get the call state by subscribing to notifications (especially the STATUS notifications)*
***Error handling**: No exceptions are thrown from the API. For error handling check the returned boolean values and the check the notifications or state query related functions to detect SIP session issues such as failed registrations or call disconnects.*
***Threading**: The API is thread-safe and most of the API functions doesn't block (unless explicitly mentioned).*

The **line** parameter is part of most of the functions and it means the channel number to be used if you wish to select a specific session.
For simple use cases usually you just have to set it to **-1**. For the details, see the "Line parameter" and the "Multiple lines" FAQ points.

Function string parameters can be passed in encrypted format. (Read the FAQ for more details regarding the encrypted parameters)

boolean webphone()

JVoIP constructor.
Create a webphone class object instance. This class contains the JVoIP public API functions.
Optionally you might also pass a startsipstack parameter.
If the set to 0, then the SIP stack will not start automatically and you will need to use the API_Start() function to start it.
This is useful to prevent loading the old/cached settings before you set any new settings.

Example:

```
webphone wobj = new webphone();  //might auto start if previous parameters were cached
webphone wobj = new webphone(0); //do not auto-start
```

boolean API_SetParameter(String param, String value)

Configure JVoIP: param is the parameter name (key). The value can be string, int and boolean.
Most of the parameters can be set with this function except gui parameters (like the colors).
If you call this function after init/start, then some parameters can take effect only when JVoIP is reinitialized.
See the full list of supported parameters here.
Example: `wobj.API_SetParameter("loglevel", 5);`

Note: some advanced parameters can be set also by line using the API_SetLineParameter function (for advanced users only; use the other API functions instead to control the individual lines). Parameters can be read with the String API_GetParameter(String param) and String API_GetLineParameter(int line, String param) functions.

boolean API_SetParameters(String parameters)

Pass a set of parameters with this function in value=key lines separated by CRLF (\r\n).
It has the same effect like the API_SetParameter function, but uses a list of parameters at once instead of setting them individually.
See the full list of supported parameters here.
Example: wobj.API_SetParameters("loglevel=5\r\ndtmfmode=1\r\n");

boolean API_SetCredentials(String server, String username, String password, String authname, String displayname)

Will set the SIP server address (ip:port or domain:port) the SIP username and the password. These values can also be preset by parameters.
Function parameters with empty strings will be omitted. For example if you would like to change only the username and the password, you can write API_SetCredentials(“”,”newusername”, “newpassord”)
If authname is empty, then the username will be used for authentications. The displayname is usually empty (no special displayname will be presented for peers). If other parameters are empty, then they can be specified by user input (If the VoIP SDK has a visible user interface).

boolean API_SetCredentialsMD5(String server, String username, String md5, String realm)

Instead of passing the password directly you can use MD5 checksum.
In this case the md5 parameter must be the md5 checksum for username:realm:password

The realm parameter is optional (can be set as an empty string) but it is recommended for easier error detection. If present and the server realm don’t match with this one, an error message will be displayed by JVoIP.

boolean API_Start()

This has to be called only if you use JVoIP as and SDK or as a java voip library. Don’t call it from JavaScript. Will start the engine.
The autostart behavior can be altered with the startsipstack setting.

boolean API_StartStack()

This function call is optional to start the sip stack on demand.
If not called, then the sip stack is started anyway if the startsipstack parameter is set, otherwise will start at first registration or outgoing call attempt.

boolean API_Register(String server, String username, String password, String authname, String displayname)

Will connect to the SIP server.
Note: Registration can be also performed automatically at startup depending on the startsipstack and register parameters.
Parameters can be empty strings if you already supplied them by parameters or by the API_SetCredentials call.
If you already passed the server, username and password (or md5) parameters with the API_SetCredentials functions, then you can call this function with empty parameters: API_Register(“”,””,””,””,””);
This function have to be called only once at the startup. Further re-registrations are done automatically based on the registerinterval parameter.
If called multiple times than the old registrar endpoint is deleted and a new one will be created with a new call-id and JVoIP will reregister.
Even if you wish to force re-registration, you should not call this more frequently than 40 seconds (because up to 40 seconds might be needed for a slow registration attempt especially if tunneling is used or if JVoIP has to negotiate the transport protocol or the register expires interval).

boolean API_RegisterEx(String accounts)

You can use this function for one ore more secondary accounts (up to 99) on the same or other servers.
Multiple accounts can be passed at once separated by semicolons (;).
An account must be specified as a SIP URI or as the following parameters separated by comma (,) :
Accounts parameters:

- Server address
- Username
- Password
- Register interval
- SIP proxy
- SIP realm
- Auth user name (if separate extension id and authorization username have to be used)
- Displayname
- Transport protocol (default/empty, UDP, TCP, TLS)

The Username parameter is mandatory, all the others are optional. JVoIP will use the defaults for the empty parameters.
All accounts have to be passed in a single line which should look like this: server,usr,pwd,ival;server2,usr2,pwd2,ival2;etc…

Note:
When you call the API_RegisterEx function, these accounts will be remembered (like you would set them with the extraregisteraccounts parameter) and will be re-used also at next startup. You can clear the old accounts by passing “null” as the accounts parameter.

Examples:

```
wobj.API_RegisterEx("user:pwd@domain:port"); //add a single extra account as a SIP URI
wobj.API_RegisterEx("sip:john:secret@myserver.com:5060"); //add a single extra account as a SIP URI
wobj.API_RegisterEx("\"John Smith\" <john:secret@myserver.com:5060>"); //add a single extra account as a SIP URI specifying also the displayname
wobj.API_RegisterEx("myserver.com,john,secret,180"); //add a single extra account as parameters
wobj.API_RegisterEx("myserver.com:5061,john,secret,180, , , , ,TLS"); //add a single extra account with TLS transport
wobj.API_RegisterEx("john:jsecret@192.168.1.50:5060;kate:ksecret@192.168.1.50:5060"); //add multiple account as SIP URI's
wobj.API_RegisterEx("92.168.1.50:5060,john,jsecret,180;92.168.1.50:5060,kate,ksecret;92.168.1.50:5060,mary,msecret,600"); //add multiple account as parameters with/without register interval
wobj.API_RegisterEx("null"); //clear old settings
```

See the Multiple account registration FAQ for more details.

boolean API_Unregister()

Will stop all endpoints (hangup current calls if any and unregister).

If you wish also the pending calls to be disconnected with unregister, set the disconunregister parameter to 1, otherwise set it to 0.
Other behaviors can be modified with the waitforunregister and clearcredentialsonunreg parameters.

Extended version:

There is also a longer version to specify other parameters.
boolean API_Unregister(String account, String server, int waitfor, int stopallsessions)

You might use this function for example if you have multiple accounts (set by the API_RegisterEx function or with the extraregisteraccounts parameter as described here) and you wish to unregister only one single account.

Possible parameters:

- account: username or SIP URI or Call-ID (in case if you have multiple/extra registered accounts)
- server: SIP server address (optional. might be used in case if you have accounts across multiple servers)
- waitfor: -1 (auto), 0 (do not wait) or 1 (wait for unregister to complete before to return)
- stopallsessions: -1 (auto), 0 (only unregister) or 1 (disconnect all other sessions belonging to this account)

Examples:

```
wobj.API_Unregister(); //unregister and stop all endpoints (including hang-up of any ongoing calls)
wobj.API_Unregister(1); //same as above but it will attempt to wait for competition up to 2000 milliseconds (modifiable by the waitforunregister parameter)
wobj.API_Unregister("2222"); //unregister one account. The account can be a username, a SIP URI or a SIP Call-ID
wobj.API_Unregister("2222@example.com"); //unregister the 2222@example.com account
wobj.API_Unregister("2222", "example.com"); //unregister the 2222@example.com account
wobj.API_Unregister("2222", "", -1, 1); //unregister the 2222 account and will also stop all related sessions (hung-up calls, unsubscribe, etc)
```

boolean API_CheckVoicemail(int line)

Will (re)subscribe for voicemail notifications. No need to call this function if the voicemail parameter is set to 2.
The line parameter should be set to -1.
See the MWI notification about incoming message waiting indicators.

boolean API_SetLine(int line)

Will set the current channel. (Use only if you present line selection for the users. Otherwise you don't have to take care about the lines).
Note: Instead of using each API call with the line parameter, you can just use this function when you wish to change the active line and use all the other API calls with -1 for the line parameter. See the Multiple Lines FAQ point for more details.

int API_GetLine()

Will return the current active line. This should be the line which you have set previously except after incoming and outgoing calls (the SIP client will automatically switch the active line to a new free line for these if the current active line is already occupied by a call)

string API_GetLineDetails(int line)

Get details about a line. The line parameter can be -1 (will return the "best" line state).
Will return the following string:
LINEDETAILS,line,state,callid,remoteusername,localusername,type,localaddress,serveraddress,mute,hold,remotefullname

Line line number (might be useful if you pass -1 as line parameter
State same as in STATUS notification
CallID: SIP session id (SIP call-id)

Remoteusername is the other party username (if any)
Localusername is the local user name (or username).
Type is 1 from client endpoints and 2 from server endpoints.
Localaddress: local IP:port
Serveraddress: remote IP:port
Mute: is muted state. 0=no,1=undefined,2=sendonly,3=recvonly,4=both muted (or if you wish to simplify: 0 means no, others means yes)
Hold: is on hold state: 0=no,1= undefined,2=sendonly,3=recvonly,4=both in hold (or if you wish to simplify: 0 means no, others means yes)
Remotefullname is the other party display name if any

boolean API_Call(int line, String peer, int calltype)

Initiate call to a number or sip username.
If the peer parameter is empty, then will redial the last number.

The line should be set to -1 usually (JVoIP will use the next free line in this case).
If line is positive, then will try to use this line number for the call if it is free.

The calltype parameter is optional and it can have the following values:

- -1: auto (default)
- 0: initiate voice call
- 1: initiate video call
- 2: initiate screensharing session

Example: `wobj.API_Call(-1, "DESTINATION");`

Note:

- *In case if your SIP server (set by the serveraddress parameter) is responsible to handle the call to the target number/extension/user, then you should pass only the peer number, extension or username to this function and not the full SIP URI (the SIP stack will construct the full SIP URI with the serveraddress already set).*
- *DTMF digits can be also appended as described here.*

boolean API_Hangup(int line, String reasontext)

Disconnect current call(s). If you set -2 for the line parameter, then all calls will be disconnected (in case if there are multiple calls in progress). The "reasontext" parameter is optional.

boolean API_Accept(int line, int calltype)

Connect incoming call.
You should call this function when there is an incoming ringing call if you wish to connect the call.
The calltype parameter accepts the following values:

- -1: default
- 0: audio only
- 1: with video
- 2: force video

boolean API_Reject(int line)

Disconnect incoming call. (API_Hangup will also work)

Certain disconnect reason codes can be specified by the following parameters:

- *disccode: the default disconnect code. Defaults to 480.*
- *userrejectdisccode: if the user rejects the cal. Defaults to 603.*
- *blockrejectcode: if call is blocked by white-list or black-list. Default is 403.*

boolean API_Forward(int line, String peer)

Forward incoming call to peer (with 301 or 302 disconnect code).
More details here.

boolean API_Transfer(int line, String peer)

Transfer current call to peer which is usually a phone number or a SIP username. (Will use the REFER method after SIP standards).

You can set the mode of the transfer with the transfertype parameter.
If the peer parameter is empty than will interconnect the currently running calls (should be used only if you have 2 simultaneous calls)
This function should be called after call connect as most SIP servers doesn't support REFER before call connect.

More details about call transfers can be found here.

boolean API_TransferDialog()

Instead of calling the API_Transfer function and pass a number, with this function you can let JVoIP to ask the C number from the user.

boolean API_AddVideo(int line, int calltype)

Add video media for an existing voice call.
The calltype can have the following values:

- 1: add video
- 2: add screensharing

More details here.

boolean API_StopVideo(int line)

Will stop the video stream at the specified line.

boolean API_Mute(int line, boolean mute, int direction)

Mute current call.
The line parameter is the endpoint. -2 for all or -1 for current line.
Set the mute parameter to true for mute or false to un-mute.
The direction can be set to one of the followings:

- 0: mute in and out
- 1: mute out (will mute the speakers, so the local user will not hear anything)
- 2: mute in (will mute the microphone, so the other party will not hear anything)
- 3: mute in and out (same as 0)
- 4: mute default (set by the defmute parameter, which is "mute microphone only" by default)

Note:

- The mute function should be used only for endpoints in call and it will be ignored if not in call
- If you wish the audio to be always muted, then you might set the defsetmuted parameter to 1 and use the default direction.
- If you don't wish to use audio device at all, then you might set the useaudiodeviceplayback and/or useaudiodevicerecord parameters to false.
- You can also change the automute parameter to automatically mute other lines on new calls.
- If you wish to keep sending RTP packets (with silence) while muted, then set the sendrtponmuted parameter to true
- For more details see here.

int API_IsMuted(int line)

Return if the selected line is muted or not.
Return values:

- -1: unknown
- 0: not muted
- 1: both muted (in/out)
- 2: out muted (speaker)
- 3: in muted (microphone)
- 4: both muted (in/out; same as 1)

int API_IsOnHold(int line)

Query if the selected line is on hold or not
Return values:

- -1: unknown
- 0: no
- 1: not used
- 2: send only (JVoIP will suppress playback)
- 3: receive only (JVoIP will suppress recording)
- 4: both in hold

Note: pass -2 for the line to find if any endpoint is in hold

boolean API_Hold(int line, boolean hold)

Hold current call. This will issue an UPDATE or a reINVITE.
Set the second parameter to true for hold and false to reload.

Hold means that the media stream is muted in one direction or both directions.
You can modify the holdtype parameter after your needs (also at runtime, by changing the holdtype parameter just before calling API_Hold).
This function should be used only after call connect as many SIP servers doesn't support UPDATE/re-INVITE before call connect.

boolean API_HoldChange(int line)

Same as API_Hold, but without the second parameter. This call will always invert the hold state for an endpoint (If the call was active, then it will switch to held state and if the call was in hold, then it will reactivate it).

boolean API_Conf(String peer)

Simply add people to conference.
If peer is empty than will mix the currently running calls (if there is more than one call)
Otherwise it will call the new peer (usually a phone number or a SIP user name) and once connected will join with the current session.

Example:

```
wobj.API_Conf("user_to_add"); //add user to conference
```

JVoIP is capable for both 3 way SIP conferencing and also to create and handle conferences with unlimited number of participants using its own built-in RTP mixer. Conference parties can use various codec's, including both 8kHz narrowband and 16 kHz wideband.

boolean API_ConfEx(int line, String peer, boolean add, int confline)

Add/remove people or line to conference.
Parameters:

- Line: channel number to be created/added or removed (usually set to -1 if peer is set)
- Peer: new username or phone number to be called and added to conference
- Add: true for new conference calls. False to remove line/number from conference.
- Confline: existing line in conference (usually set to -1)

If peer is empty string than:
-if add is true:
-if line is -2 then it will mix all the currently running calls (if there is more than one call)
-if line is not -2, then it will add the channel (to the existing call/conference)
-if add is false:
-if line is -2 then it will destroy the conference (but will keep the calls on individual lines)
-if line is not -2 then it will remove the selected line from the conference
Otherwise it will call the new peer (usually a phone number or a SIP user name) and once connected will join with the current/old call session.
If line is set to a positive number then it will try to use that line number for the new call (if it is free).

The confline is an optional advanced parameter which should be used only if you wish to create multiple simultaneous conference calls.
If set to a positive number, then the new conference party (specified with the line and/or peer parameter)
will be added (join) to the call or conference where the confline belongs.

Examples:

```
wobj.API_ConfEx(-1, "user_to_add", true); //add user to conference
wobj.API_ConfEx(2, ""); //add line 2 to conference
wobj.API_ConfEx(-2); //destroy all conferences (remove all endpoints from conference calls)
wobj.API_Hangup(-2); //disconnect all calls
wobj.API_ConfEx(2, "", false); //remove line 2 from conference
wobj.API_ConfEx(8, "", true, 4); //add line 8 to conference on line 4
```

Please note that to be able to make conference calls, you should make one (or two) call first using the API_Call function.
Then use the API_Conf or API_ConfEx to convert the existing call to a conference and to add one or more peers.

You might need to set the conference_addalllines parameter to 0 if you wish to explicitly/strictly control the conference lines.
You might set the conference_tracklines parameter to 1 if you wish to use multiple simultaneous conference calls.

boolean API_Dtmf(int line, String dtmf)

Send DTMF message by SIP INFO, RFC2833 or In-Band method (depending on the “dtmfmode” parameter). Please note that the dtmf parameter is a string. This means that multiple dtmf characters can be passed at once and the VoIP SDK will streamline them properly. Use the space char to insert delays between the digits.

Valid dtmf characters in the passed string are the followings: 0,1,2,3,4,5,6,7,8,9,*,#,A,B,C,D,space.

The dtmf messages are sent with the protocol specified with the dtmfmode parameter.

Feedback about the message delivery can be received with the INFO notifications.

Example: API_Dtmf(-2,” 12 345 #”);

Note: dtmf messages can be also sent by adding it to the called number after a comma. For example if you make a call to 123,456 then it will call 123 and then it will send dtmf 456 once the call is connected.

For more details see the How to DTMF FAQ point.

boolean API_Info(int line, String msg)

Send any SIP INFO message as defined in RFC 2976.

Use this API to send any custom INFO messages to the server or to the connected peer (in case if your server will forward it to other peer and not discard it).

The msg string will be sent in the INFO body.

The Content-Type can be specified with the contenttype parameter. For example you might set the contenttype to “application/mydata”. If the contenttype is not set then it will be auto guessed as application/json, application/xml or application/octet-stream”

Feedback about the message delivery can be received with the INFO notifications.

Example: API_Info(-2,”MyMessage”);

For more details see the Custom INFO messages FAQ point.

boolean API_SendUSSD(int line, String method, String ussd)

Send an USSD message after the IMS 3GPP TS 24.390 standard.
The method parameter can be set to “INVITE”, “INFO” or “BYE”.
If INVITE, then a new session will be created. Otherwise will use the session suggested by the line parameter.
The ussd parameter have to be set to the USSD string (for example “*135#”) or whole XML (“<?xml version … “).
USSD strings should be up to 182 characters.

Will return true if send initialized successfully or false on failure.
Further feedback about the actual delivery or incoming ussd messages can be obtained from the USSD notifications.
More details here.

boolean API_SendChat(int line, String peer, String group, String message, int msgid)

Send a chat message. (usually SIP MESSAGE method after RFC 3428 or as configured by the textmessaging parameter)
Parameters:

- line: channel number. Usually you can just set it to -1.
- peer: phone number or SIP username/extension number.
- group: optional group name parameter. Used only for group chat, otherwise it can be an empty string.
- msgid: optional id. You can pass any (unique or auto incrementing) positive number to match the CHATREPORT notifications.

Example:

```
wobj.API_SendChat(-1, "john", "", "Test Message");
```

Check the CHATREPORT notification to see if delivery succeeded or failed.
On successful delivery you will also receive a log like: EVENT, chat sent successfully
On failed delivery you will also receive a log like: WARNING, chat message not delivered
You can also send typing notifications with the API_SendChatIsComposing function.
Incoming instant messages (IM or SMS) are reported as CHAT notifications.

boolean API_SendSMS(int line, String peer, String message, int id)

Send a SMS message if softswitch has SMS delivery capabilities (Otherwise might try to deliver as IM).

The message is delivered as a 3GPP SMS, SMS HTTP API request or a standard SIP MESSAGE with X-Sms: Yes header (in this case the server is responsible to convert it to SMS).

Everything is like API_SendChat, except that this function will try to send SMS instead of chat (also depending on the textmessaging parameter).

boolean API_Chat(String peer)

Instead of calling the API_SendChat function and pass a message, with this function you can let the voip client to open its built-in chat form.
Will open the chat dialog (the "number" parameter can be empty)
Peer can be a SIP username or extension number.
This function will not work with the headless version (the headless is for library or command line usage and it doesn't have any built-in GUI).

boolean API_VoiceRecord(int startstop, int now, String filename, int line)

Will start/stop a voice recording session.

- startstop: 0 to stop, 1 to start locally, 2 to start remote ftp or http, 3 start to record both locally and to remote ftp/http, 4 start to record as it is set by the "voicerecording" parameter
- now: used if the startstop is set to 0. 0 means that the recorded file will be saved and/or uploaded at the end of the conversation only. 2 means that the file will be saved immediately
- filename: file path and/or name used for storing the recorded voice (if empty string, than will use a default file path and name)
- line: set to -2 for global, -1 for the current active line or specify channel number

Examples:

```
//start recording any calls to local file (file will be stored as specified by the voicerecfilename and voicerecformat parameters)
wobj.API_VoiceRecord();
or
wobj.API_VoiceRecord(1, 2, "", -2);

//start recording on all lines as specified by the voicerecording, voicerecfilename and voicerecformat parameters.
wobj.API_VoiceRecord(4, 2,"", -2);

//stop call recording on all lines
wobj.API_VoiceRecord(0);
or
wobj.API_VoiceRecord(0, 2,"", -2);

//start recording the current active call and upload to the specified ftp once ready in the specified format (wav) and file name.
wobj.API_VoiceRecord(2, 2,"ftp://ftpuser:ftppassword@myftpserver.com:21/callrecord_DATETIME_USER.wav", -1);

//start recording on the current active call and upload to the specified web-service once ready (also keep the file locally)
wobj.API_VoiceRecord(3, 2," http://www.foo.com/myfilehandler.php/FILENAME", -1);

//will stop call recording on line 1 once call is terminated
wobj.API_VoiceRecord(0, 0,"", 1);
```

Notes:
This function should be used only if you would like to control the recording duration.
If all conversations have to be recorded, then just set the "voicerecording" parameter after your needs.
The recording starts when you call this function with a positive startstop parameter and will end once the call disconnects or when you call it again with startstop parameter set to 0.

This function is capable to overwrite most of the voice recording related parameters: for example if you specify a full URL as the file name, then it will use this URL instead of the URL specified by the http_addr or voicerecftp_addr parameter. If you set a file name terminating with a file extension (such as .wav or .mp3) then it will recording in this format instead of the format preconfigured by the voicerecformat parameter.

For more details see the voice record FAQ.
(*In case if you wish to receive the audio packets/audio stream in real time, then you can use the local audio streaming capabilities instead)*

boolean API_PlaySound(int start, String file, int looping, boolean async, boolean islocal, boolean toremotepeer, int line, String audiodevice, boolean isring)

Play any sound file.
This function can be used to play audio locally, but also for remote streaming.

The file must be found near JVoIP.jar.

- start: 1 for start or 0 to stop the playback, -1 to pre-cache
- file: file name
- looping: 1 to repeat, 0 to play once (only for local playback)
- async: false if no, true if playback should be done in a separate thread (false can be used only for local playback, not for streaming)
- islocal: true if the file have to be read from the local machine file system. False if remote file (for example if the file is on the webserver)

- toremotepeer: stream the playback to the connected peer
- line: used with toremotepeer if there are multiple calls in progress to specify the call (usually set to -1 for the current call if any)
- audiodevice: you can specify an exact device for playback. Otherwise set it to empty string
- isring: whether this sound is a ringtone/ringback

Examples:

-playback a file locally (mysound.wav must exist near the JVoIP.jar file):
API_PlaySound(1, "mysound.wav", 0, false, false, false, -1,"",false)
-playback a file to the connected remote peer (mysound.wav must exist near the JVoIP.jar file):
API_PlaySound(1, "mysound.wav", 0, false, true, true, -1,"",false)
-stop the playback:
API_PlaySound(0, "", 0, false, false, false, -1,"",false)

Notes:

- *The singleaudiostream parameter must be set to 0 if you wish to play simultaneous streams to remote (for multiple concurrent calls)*
- *You might set the useaudiodevicerecord parameter to false if you need streaming but don't have an audio recorder device installed.*
- *If you plan to use this functionality running JVoIP on a server, then have a look at the Using JVoIP on a server FAQ*
- *If the previous sound file haven't finished the playback yet when you supply a new one then the remaining from the old one will be dropped and the new file will start to be played immediately*

File format:

At least wave files (raw linear PCM) are supported in 8kHz 16 bit mono format: PCM SIGNED 8000.0 Hz (8 kHz) 16 bit mono (2 bytes/frame) in little-endian (128 kbits). In case if your wave file is narrowband (8 kHz 16 bit mono), then the stream will be automatically converted to wideband if the VoIP audio codec is wideband such as opus or speex. In case if your wave file is wideband (16 kHz 16 bit mono) then you should set the playbackstreamformat parameter to 1 to allow auto conversion to narrowband if the VoIP audio codec is narrowband such as G.711 (PCMU, PCMA), G.729 or GSM.

boolean API_StreamSoundBuff(int start, int line, byte[] buff, int len)

Stream from raw wave audio PCM (Linear PCM) buffer or from RTP packets.
This function will stream the supplied audio buffer to the peer endpoint, transcoding if necessary.

Parameters:

- start: 1 for start or 0 to stop the playback
- line: specify the channel in case if there are multiple calls in progress (usually set to -1 for the current call if any)
- buff: audio buffer (raw PCM data or RTP packet)
- len: length of the buff (ideally it should be multiple of 360 if raw pcm or the size of the RTP packets which are to be passed separately). If len is 0 or negative and start is 1, then len will be set from buff.length.

Notes:

- *The buff should not contain any file header if you are using raw PCM (only raw linear PCM audio data) and it should contain a full RTP packet (with the RTP header) if you are passing RTP data.*
- *You might set the useaudiodevicerecord parameter to false if you need streaming but don't have an audio recorder device installed.*
- *If previously supplied buffer(s) are not completed yet, then the new one will be queued.*
- *If you are streaming in real-time (feeding the API with short RTP/audio packets) then you might set the realtimeplayback parameter to 1.*
- *Alternatively you can use the API_StreamSoundStream function to pass an InputStream instead of byte buffers.*
- *In case if you wish to send (also) video, then use the API_SendVideoRTP function.* See the video guide and the video streaming guide for more details.
- *In case if you wish to receive the audio stream from the remote peer endpoint, see this FAQ point instead.*

Buffer/stream format:

At least raw linear PCM is supported in 8kHz 16 bit mono format: PCM SIGNED 8000.0 Hz (8 kHz) 16 bit mono (2 bytes/frame) in little-endian (128 kbits). In case if your buffer is narrowband (8 kHz 16 bit mono), then the stream will be automatically converted to wideband if the VoIP audio codec is wideband such as opus or speex. In case if your buffer is wideband (16 kHz 16 bit mono) then you should set the playbackstreamformat parameter to 1 to allow auto conversion to narrowband if the VoIP audio codec is narrowband such as G.711 (PCMU, PCMA), G.729 or GSM. With raw PCM the supplied buffer length should be a multiply of 160 or 320, otherwise extra bytes might be dropped. The function can accept also RTP packets and will transcode them if necessary. Set the streamsoundisrtp parameter to -1 to auto-guess raw PCM vs RTP (this is the default value), set to 0 if you supply raw PCM buffer or set to 1 if you supply RTP packets.

See more details here.

byte[] API_GetMedia(int timeout)

Retrieve the next media (audio or video) packet as a byte buffer from the JVoIP internal queue.
The buffer format (prefix bytes) can be specified with the related sendmedia_... parameters as described in the streaming FAQ.

This is a blocking function call, so you should call it from a separate thread.
The timeout can be specified in milliseconds. If it is higher then 0, then JVoIP will use an additional waiting lock. If 0 then will issue a simple/fast blocking read.
The internal queue size is 900 and on overflow it will remove all queued packets (if you are not calling this function or not fast enough).

The internal media packets queueing will start only if the sendmedia_mode is set to 2 or 3 or at first call to this function.
(The first call for this function will also set the sendmedia_mode parameter to 2 or 3 if it was not set before)
If you don't need media streaming anymore, then you might call the API_GetMediaEnd() function to stop queuing any more media packets, otherwise it will stop automatically after some time of no usage or overflow.

The receive byte buffer might contain some header bytes as specified by the sendmedia_ parameters.
For more details see the streaming FAQ.

boolean API_AudioDevice()

Open audio device selector dialog (built-in user interface).
This function will not work with the headless version (the headless is for library/SDK or command line usage and it doesn't have any built-in GUI).

string API_GetAudioDeviceList(int dev)

Will return the list of available audio devices separated by \r\n.
Dev parameter
0: list the recording (audio in / microphone) device names
1: get the playback/speaker or ringer devices
2: list all devices
Note:
Instead of using this function, you might just call the "API_AudioDevice" to let the users to change their audio settings.
When using the native winapi audio engine, the device names might be truncated to the first 30 character due to the wave audio API limitations.
You can pass the same back to the webphone and it will select the audio device correctly.

string API_GetAudioDevice(int dev)

Will return the currently selected audio device for the dev line (dev values are 0 for recording, 1 for playback, 2 for ringer).

boolean API_SetAudioDevice(int dev, string devicename, int immediate)

Select an audio device. The devicename should be a valid audio device name (you can list them with the API_GetAudioDeviceList call)
The "dev" parameter can have the following values:
0: recording device (microphone)
1: playback device (speaker, headset)
2: ringer device (speaker, headset)
The "immediate" parameter can have the following values:
0: default (after the "changeaudiodevimmediate" parameter)
1: next call only
2: immediately for active calls
Note:
For the ringer device you can also pass the "All" string as the devicename to make voip application to ring on all devices for incoming calls.
All devices can also accept the "Default" device name, which will select the system default audio device.
For the device you can also pass a number which is the order of the audio device as returned by API_GetAudioDeviceList staring from 1. Valid values are between 1 and 9 or 0 for the system default device.
This function internally will set the audiodevicein / audiodeviceout / audiodeivering parameters.
Instead of this function you might just call the "API_AudioDevice" function which will present an user interface to let the users to change their audio settings.

boolean API_SetVolume(int dev,int volume, int line)

Set volume (0-100%) for the selected device.
The dev parameter can have the following values:
0 for the recording (microphone) audio device
1 for the playback (speaker) audio device
2 for the ringback (speaker) audio device
The line is an optional parameter to be used only if you need different volume for separate calls.

Note: The ringer volume might not be always honored and might not change during ring playback.
This can be influenced also with the toneplayermode parameter: -1: auto/default, 0: force clip playback, 1: use audioplayer of device is set, 2: use audioplayer also if ringtone is set, 3: always use audioplayer.

int API_GetVolume(int dev, int line)

Return the volume (0-100%) for the selected device.
The dev parameter can have the following values:

0 for the recording (microphone) audio device
1 for the playback (speaker) audio device
2 for the ringback (speaker) audio device

The line is an optional parameter to query the volume of a specific call line in case if you set the volume per call. Otherwise don't use or set to -2.

String API_VAD(int line)

Returns voice activity statistics.
It will return the VAD state string as described at the VAD notification.
If line is 0 or negative then it will report global send/recv statistics.
If line is positive then it will report receiver state for the requested line.
You should set the vad parameter to at least 2 (4 for full report), although calling this function will also set it, but at first call will not have statistics if it was not set.
Note: if you call this function, VAD notifications will not be sent automatically anymore. (So you will need to continue to poll for the details).
More details here

String API_RTPStat()

Returns media statistics. See the RTPSTAT notification for more details.
For this to work you should set the vad parameter to at least 2 (4 for full report).
Note:
RTP statistics can be sent also automatically if you set the rtpstat parameter.
If you call this function, RTPSTAT notifications will not be sent automatically anymore
More details here.

String API_GetVersion()

Return the program version number.

String API_GetStatus(int line, int strict)

Returns line state or global state if you pass -2 as line parameter. The possible returned texts are the same like for STATUS notification strings.
If the strict variable is set to 1, then it will return "Unknown" if no such line is activated. If the strict variable is set to 0, then it will return the default active line if the line doesn't exists.
You should use the notifications described below to get the actual state of JVoIP instead of continuously polling it with this function call.

boolean API_SetNotificationListener(SIPNotificationListener listener)

Subscribe to notification events.
Create a subclass of the JVoIP SIPNotificationListener first, then pass this object as a listener.
You will receive the SIPNotification objects overriding the member functions.
More details:

- Notifications
- Javadoc
- Code template
- Example code

String API_PollNotificationStrings ()

Should be used only if you wish to receive the notifications as strings.
Use the API_SetNotificationListener() instead to receive SIPNotification objects instead of strings.

Return the notification strings. You should poll for the notifications periodically from a separate thread. It will return accumulated events since the last function call (notifications separated by \r\n -CRLF).
You might set the "polling" parameter to 3 if you wish to use this function (and not socket or webphonetojs events)

More details here.

Note:

- *We are using simple polling for this since Java doesn't have function pointers to be used as a callback and other methods are either deprecated (Observer interface) or too complex (external libraries).*
- *Once a notification string have been read then it will be cleared from the internal list, so it is guaranteed that you will never receive duplicates.*
- *If you are using the library from javascript then just use a timer instead of a thread*

- *The old function name as API_GetNotifications which was renamed for more clarity (the old way will still be kept)*

String API_GetNotificationStrings ()

Should be used only if you wish to receive the notifications as strings.
Use the API_SetNotificationListener() instead to receive SIPNotification objects instead of strings.

Same as above API_PollNotificationStrings(), but will blocking wait for data (more efficient) and it should be used from a separate thread.
The old function name as API_GetNotificationsSync which was renamed for more clarity (the old way will still be kept)

SIPNotification API_PollNotification()

Use the API_SetNotificationListener() instead to receive the notifications as objects instead of strings.

Same as above API_PollNotificationStrings(), but will return a SIPNotification object instead of string.

SIPNotification API_ParseNotification (String notificationstring)

Convert a notification string to a SIPNotification object.
Useful only if you receive the notifications as strings (instead of SIPNotification objects which is recommended).
More details here.

Contacts

Contacts are normally handled by the caller process (your app) because they have less to do with the VoIP stack (except presence state). This should be done easily from your application and better adopted to your needs. You can use the above JVoIP API to add VoIP functionality to your address book or contact list. The state (Online/Offline/Busy) of the users can be loaded from your softswitch database. Based on the user presence you can display the different buttons with your design. Near each contact you can display a call/chat button which will launch a voip endpoint instance preconfigured with the actual contact ("callto" parameter).

However for your convenience, the VoIP SDK also provides a simple contact management API.

Contact parameters are stored as comma delimited strings with the following parameters:
imstatus,name,sip,phone,phone2,phone3,othernumbers,sipcontacturi,email,web,address,speeddial,extra,internalextra

boolean API_SetContacts(String contacts)
Set all contacts (contact parameters separated by new line)

String API_GetContacts()
Will return all contacts (in separated lines the parameters described above)

boolean API_DelContact(String name)
Delete contact.

boolean API_AddContact(String params)
Add a contact. Example: Add_Contact(',John Smith,jsmith,'); //here we set only the name and the SIP fields

boolean API_SetContact(String name, String params)
Change contact.

String API_GetContact(String name)
Will return a single contact in the format described above.

Helper functions to set/get individual fields:

boolean API_SetContactName(String name, String param)
Set contact name.

boolean API_SetContactSIP(String name, String param)
Set contact sip uri.

boolean API_SetContactPhone(String name, String param)
Set contact phone number.

boolean API_SetContactSpeedDial(String name, String param)
Set contact speed dial number (short number).

String API_GetContactName(String name)
Get contact name.

String API_GetContactSIP(String name)
Get contact sip uri.

String API_GetContactPhone(String name)
Get contact phone number.

String API_GetContactSpeedDial(String name)
Get contact speed dial number (short number).

Presence

Presence is based on SIP SIMPLE SUBSCRIBE/NOTIFY mechanism and it is used to detect the online status of the contacts.
There is no need to manage the contacts within SIP endpoint (as described above) to have presence functionality (so you can manage the contacts externally in your application).
The following steps are required:

1. Related settings:
    - enablepresence: 0/1
    - email = email address sent with contact info
    - presenceexpire = 3600
    - autoacceptpresencrequests = -1; //-1: not set (1), 0=auto reject all,1=ask for new users,2=yes, autoaccept new unknown users
2. First you should call API_PushContactlist to pass all the usernames and phonenumbers from your external contact list if any. This is necessary, because for existing contacts JVoIP can accept the requests automatically, while for other it might ask for user permission
3. On first start you might call API_NumExists. If using the Mizu VoIP server, then it will return all existing contacts with SERVERCONTACTS,userlist notification where userlist are populated with the valid users and their online status.
4. Call API_CheckPresence(userlist). To save softswitch resources, you should carefully select the contacts. (Send only the contacts which are actually used and called numbers. We recommend up to 50 contacts. If the user select a contact, then you can call this function later with that single contact to request its status). *This function will start to send SUBSCRIBE requests.*
5. Use the API_SetPresenceStatus(statustring) function call to change the local user online status with one of the followings strings: Online, Away, DND, Invisible , Offline (case sensitive) . *This function will start to send NOTIFY requests to subscribed parties.*
   Note: presence per user can be also set by upresence_USERNAME parameters.
6. Once these are done, the following notifications can be received from java sip stack:
   NEWUSER,peerusername,displayname,email,URI
   PRESENCE,peerusername,status,details,displayname,email
   (displayname and email can be empty)

   On newuser, you should ask the user if wish to accept it. If accepted, call the API_NewUser function. The same function should be called when the user adds a new contact to its contactlist.
   For presence the following status strings are defined (be prepared to receive any of these and handle it with case insensitive by displaying red/green/gray/other icons):
    - PRESENCE_ONLINE: Open/Online/Reachable/Available/Call Me/Registered [GREEN]
    - PRESENCE_DND: DND (Do not disturb; halt popups and sounds) [RED]
    - PRESENCE_BUSY: Busy/Speaking (can be auto set) [ORANGE/YELLOW]
    - PRESENCE_PENDING: Pending/Forwarding [ORANGE/YELLOW]
    - PRESENCE_AWAY: Away/Idle [ORANGE/YELLOW]
    - PRESENCE_OFFLINE: Close/Unreachable/Offline/Unregistered [GREY/WHITE]
    - PRESENCE_UNKNOWN: Unknown/Not Set/NotExists [GREY/WHITE/NOCOLOR]
    - Invisible (no status notifications will be sent) [GREY/WHITE/NOCOLOR]

   Other suggestion for colors:
    - red: busy/dnd
    - bright green: online
    - pale green: away/forwarding/pending
    - white: user exists but unknown status or invisible
    - no color: user doesn't exists / no presence feature

7. You might use the API_UnSubscribe() API to unsubscribe all endpoints (this includes presence, voicemail and BLF subscribes).

Note: presence is initiated automatically with called contacts after call disconnect

BLF

BLF (Busy Lamp Field) can be used to monitor the state of an extension and it is implemented as SUBSCRIBE/NOTIFY with dialog event package as described in RFC 3265 and RFC 4235.

The main difference between presence and BLF are the followings:

- *They are implemented differently, based on different RFC's (both based on SUBSCRIBE/NOTIFY, but different message exchange.*
- *Presence is focusing for availability (online/offline), BLF is focusing on call state (ringing/speaking)*
- *BLF usually have to be enabled explicitly to work with selected peers, presence might be enabled by default for all "friends"*
- *Presence can be used to track friend's online state, while BLF is used mostly in offices to track another device call-state such as a secretar tracking it's boss phone to know when it is available/off-hook (for example if it is available to transfer a new call to it).*

Set the enableblf parameter to enable/disable BLF. The following values are defined:

0: disable BLF
1: auto (from here it will auto switch to 0 or 2 regarding the circumstances –whether BLF was initiated and succeed/failed)
2: enable BLF
3: force always (if you set to 3 then it can't be switched off later and will use BLF even after failure)
Default value is 1.

To subscribe to other extensions call state, you can set the blfuserlist parameter or use the API_CheckBLF(userlist) to subscribe to other extension(s) state changes (users separated by comma). To remove an extension, just modify the blfuserlist parameter or call the API_DisableBLF(userlist) API. If the userlist is an empty string then BLF for all users will be disabled.

Make sure that the peer extension(s) also has BLF support (so it can respond with NOTIFY for the BLF SUBSCRIBE).
Once the state of the remote extension(s) are changed, you will receive BLF notifications in this string format:

BLF,peerusername,direction,state,callid

SIPNotification.BLF members:

- **getPeer()**: extension username
- **getDirecton()**:
  - DIR_UNDEFINED (not for calls or not announced by the peer)
  - DIR_OUT (outgoing call)
  - DIR_IN (incoming call)
- **getStatus()**: blf state:
  - STATUS_TRYING (call connect initiated)
  - STATUS_PROCEEDING (call connecting)
  - STATUS_EARLY (ringing or session progress)
  - STATUS_CONFIRMED (call connected)
  - STATUS_TERMINATED (call disconnected)
  - STATUS_UNKNOWN (unrecognized call state received)
  - STATUS_FAILED (BLF subscribe or notify failed)
- **getStatusText()**: blf state as string
- **getCallid()**: optional value if reported from the remote extension (the SIP call-id of the call)

See the SIPNotification.BLF in the javadoc for the SIPNotification object details.

Miscellaneous

Some other not so important API calls are listed below:

boolean API_Test()
You might use this function to check the API availability. Should return true.

int API_TestEx()
You might use this function to check the API availability. Should return 42.

boolean API_Test()
You might use this function to check the API availability (should return true).

boolean API_HTTPKeepAlive()
This is needed only if used from web via JavaScript.
You should call this function periodically more frequently than the timeout specified by the "httpsessiontimeout" parameter. (For example call this in every 5 minute). This is to prevent orphaned JVoIP instances (when your html page was closed or crashed but JVoIP is still running in the background).
If you don't wish to call this function periodically, then you should set the "httpsessiontimeout" parameter to 0.

boolean API_ServerInit(String address) -deprecated
Call this function before to start any communication with this address (usually an IP number). This is required to release the Java security restrictions. Wait 1-2 second before calling the next function like API_Register or API_Call. This function is deprecated since v.3.8 (no need to call this, just call API_Register or others directly)

boolean API_StartGUI()
Launch the built-in simple phone user interface even if the app was launched from command line.

boolean API_Stop()

Will stop all endpoints. This function call is optional when you unload JVoIP from external app.

boolean API_Exit()
Will stop all endpoints and terminates the java sip application. This function call is optional when you unload the JVoIP java module or wish to issue a forced termination. Its behavior can be controlled by the “exitmethod” and “destroymode” parameters.

boolean API_CapabilityRequest(String server, String username)
Will send an OPTION request to the server. Usually you should not use this function.
The server parameter can be empty if you already set it with other API calls or by parameter.
The username parameter can be empty (in this case the “From” address will be set to “unknown”)

void API_CheckConnection()
You might call this function to quickly recover from connection failures on events not know by JVoIP such as app switched to foreground (otherwise JVoIP should auto-recover from network failures)

String API_LineToCallID(int line)
Get the SIP Call-ID for a line number.

int API_CallIDToLine(String callid)
Get the line number for a SIP Call-ID.

String API_NextCallID(String callid)
Set/get the SIP Call-ID used for the first upcoming new SIP session.
This function might be used if you need the SIP Call-ID before the call for some reason. For example you can use it before the call function to set/get the SIP Call-ID of the new call. Otherwise JVoIP will automatically generate an unique SIP Call-ID for each new session which you can query using the API_LineToCallID function or receive it with the Status or Cdr notifications.
If this function is called without the callid parameter or the callid as an empty string then JVoIP will generate and return random call-id.
The return value is the new SIP Call-ID as string.

boolean API_SetLineParameter(int line, String param, String value, int permanent)
Set parameter for the current/next/all/specific line. Only some of the parameters are supported by this function (listed below).
These are advanced settings and this function might be used only if you need some specific/different behavior for a specific line only.
Otherwise, the behavior of the different lines are loaded from the global configuration (parameters), from the circumstances
or can be influenced by the other API functions (try to use the other API’s instead of this when possible).
Parameters:

- Line: channel number.
  - If -2, then it will be set as global config like the API_SetParameter.
  - If -1, then for the active line.
  - If 0, then it will be applied for the next/new channel(s).
  - Otherwise it will be applied for the specific line.
- Param: settings key name
- Value: value as string (or "NULL" to clear any old preset)
- Permanent
  - If 1 then the setting will be kept for all future lines with the same line number
  - If 0 then the setting will be applied only for the current or very next channel with the same line

The following parameters are supported by this function:
peer (called number), address (will set both the serveraddress and proxyaddress), serveraddress, proxyaddress, username, sipusername, password, displayname, voicerecording, muted, holded, maxsipmessagesize, codecretry, disccode, registerinterval, maxmsgresend, srtp_suite, strictsrtp, dtmfmode, rtp_timestamp, rtp_seq, rtp_ssrc, defpayload, setfinalcodec, presenceexpiresec, notifyexpiresec, sipproto, volumein, volumeout, volumering, videocalltype, videodirection, keepvideosdponhold, defvideopayload, serverdomainandport, serveraddr, serverport, serverip, proxyport, proxyip, realm, p_referred_identity, p_asserted_identity, remote_party_id, privacy, customsipheader, customsdpfield, customsdpmediafield, sip_uui, sessionid, discreason, usesdpip, transport, line, callid, state, mediaencryption.

All preset settings for all lines can be removed by passing 0 for the line and empty or NULL for both the key and the value.

Examples:

```
wobj.API_SetLineParameter(1,"username", "mia",0); //set the username to mia for the next call on line 1
wobj.API_SetLineParameter(2,"username", "mia",1); //set the username to mia for all upcoming calls on line 2
wobj.API_SetLineParameter(2,"username", "NULL",1); //clear the permanently preset username from line 2
wobj.API_SetLineParameter(-1,"muted", "0",0); //unmute the current call (use the API_Mute instead)
wobj.API_SetLineParameter(0,"muted", "4",0); //set the default muted state both side mute for the next call
wobj.API_SetLineParameter(0,"transport", "1",1); //set the transport parameter to TCP for all the upcoming new sessions
wobj.API_SetLineParameter(0,"NULL", "NULL",1); //clear all the permanent settings from all lines
```

Line parameters can be read with the String API_GetLineParameter(int line, String param) function.

boolean API_SetSIPHeader(int line, String hdr)

Set a custom SIP header (a line in the SIP signaling) that will be sent with all messages. Can be used for various integration purposes (for example for sending the http session id).
Example: API_SetSIPHeader(-1, "X-MyData: VALUE");
Multiple headers can be separated by CRLF (\r\n). You can also set this with the customsipheader parameter.
Line -2 means all lines, -1 means current active line or use 1+ means for an existing line in call.
You can set a SIP header for the next call by using line number -3 or clear the old header(s) by using line number 0.

String API_GetSIPHeader(int line, String hdr)
Return a SIP header value received by JVoIP. If not found it will return a string beginning with "ERROR:" such as "ERROR: no such line".

String API_GetSDPField(int line, String field)
Return a SDP field value received by JVoIP. If not found it will return a string beginning with "ERROR:" such as "ERROR: no such line".

boolean API_SetSDPField (int line, String field, int type)
Set a custom SDP field (a line in the SDP body) that will be sent with all messages.
Set the type parameter to 0 for global values (before the m= line) or 1 for media values (after the m= line).
Example: API_SetSDPField(-1, "a=3ge2ae:requested",1);
Multiple headers can be separated by CRLF (\r\n). You can also set this with the customsdpfield and customsdpmediafield parameters.
Line -2 means all lines, -1 means current active line or use 1+ means for an existing line in call.
You can set a SIP header for the next call by using line number -3 or clear the old header(s) by using line number 0.

boolean API_RTPHeaderExtension (int line, int profile, String extension)
Set RTP header extension for the RTP packets.
The profile will be set as the first two bytes in the extension header (might be used as a custom identifier or parameter).
The extension can be one or multiple numbers separated by semicolon (;). These numbers will be sent as 32 bit words with the RTP header extension payload.
Example: API_RTPHeaderExtension (-1, 1, "987"); //will set the profile number to 1 and the rtp extension word to 987 for the current call
Line -2 means all lines, -1 means current active line or use 1+ means for an existing line in call.
You can set a SIP header for the next call by using line number -3 or clear the old header(s) by using line number 0.
Set the profile to 0 and the extension string to empty to clear it.
See the RTP header extension FAQ point for more details.

boolean API_ED137PTT (int line, int ptt, int pttid)
ED-137 Push to talk.
Parameters:
- line: channel number
- ptt: 0: off (not speaking), 1: on (Normal PTT ON), 2: Coupling PTT ON, 3: Priority PTT ON, 4: Emergency PTT ON
- pttid: optional PTT-ID (otherwise loaded from global config)

Will return true if initiated successfully or false on failure.
The ed137 parameter should be set to 1 before using this function. See the ED_137 guide for more details

boolean API_SetUUI(int line, String value, int fortarget)
Set a custom UUI (User-to-User Call Control Information as described in RFC 7433).
Value is the UUI data (with or without parameters)
Fortarget specifies if the UUI have to be sent to target party with call transfer or redirect escaped. 0: no (to peer with the User-to-User header), 1: yes (to target in Contact or Refer-To URI), 2: both.

You might call this function just before JVoIP have to send a SIP signaling message to transmit an UUI (for example before a call or just before a call transfer).

Example:
API_SetURI(-1, "anydata",-2);
API_SetURI(-1, "anydata;encoding=hex;purpose=foo;content=bar",-2);
You can also set this with the sip_uui parameter.

boolean API_SendSIPMessage(int line, String msg, String body, String account, String target)
Send a custom SIP signaling message (for example OPTIONS, NOTIFY, etc).
Parameters:
- line: endpoint where the message will be sent. -3: new endpoint, -2: all endpoints, -1: current active, 0: register endpoint, 1+: call channel number
- msg: SIP message to send. It can be just the method name (OPTIONS,MESSAGE,NOTIFY,INVITE,etc), the full message header or the full message including also a body (such as SDP)
- body: optional SIP message body (such as SDP, xml payload or chat text, etc). Set to "null" if body must not be sent.
- account: optional local SIP account details if you wish to use other parameters then the configured one. same format like the extraregisteraccounts parameter
- target: optional target username, number or SIP URI if not specified in the msg

Examples:
- Send a custom REGISTER request: wobj.API_SendSIPMessage(0, "fullregistermessagehere");
- Send UPDATE on line 1: wobj.API_SendSIPMessage(1, "UPDATE");
- Send IM on line 2: wobj.API_SendSIPMessage(2, "MESSAGE", "hi", "", "evelin");

- Send OPTIONS to another server: wobj.API_SendSIPMessage(-3, "OPTIONS", "", "username:password@otherserveraddress.com");
- Send PUBLISH with a custom body: wobj.API_SendSIPMessage(-3, "PUBLISH", "anymessagehere", "");

The Content-Type for INFO, MESSAGE and NOTIFY requests can be specified with the contenttype parameter. Otherwise the content type will be auto guessed.

String API_GetSIPMessage(int line, int dir, int type)
Return the last received or sent SIP signaling message as raw text.
Dir:
0: in (incoming/received message)
1: out (outgoing/sent message)
Type:
0: any
1: SIP request (such as INVITE, REGISTER, BYE)
2: SIP answer (such as 200 OK, 401 Unauthorized and other response codes)
3: INVITE (the last INVITE received or sent)
4: the last 200 OK (call connect, ok for register or other)

String API_GetLastRecInvite()
Return the last received INVITE message.

String API_GetLastSentInvite()
Return the last sent INVITE message.

String API_GetLastRecSIPMessage(String line)
Get the last received SIP message as clear text. Line is the line number or the SIP call id.

String API_GetLastRecFileName (String line)
Returns the last recorded file name.

String API_SendChatIsComposing (String line, String number)
Send typing notification.

boolean API_IsOnline()
Return true if network is present.
This is done with some heuristics and might detect network availability/loss only after some little time.

boolean API_IsRegistered()
Return true if JVoIP is registered ("connected") to the SIP server, otherwise false.
This is done with some heuristics as there are states when the registration state can't be determined exactly (for example when just sent a (re)register and answer not received yet or at startup or runtime failover when connection is lost and it is trying to reconnect or failover to another transport protocol)
You can also watch for Register notifications instead of (continuously) calling this API.

int API_IsRegisteredEx()
Returns extended registration state: 0=unknown,1=not needed,2=yes (registered),3=working yes (registering, probably will succeed),4=working unknown (registering), 5=working no (registering, probably will fail) ,6=unregistered,7=no (registration failed)

int API_IsInCall()
Return whether the sip stack is in call: 0=no,1=ringing,2=speaking

int API_GetCurrentConnectedCallCount()
Get number of current connected calls

String API_GetRegFailReason(boolean extended)
Will return a text about the reason of the last failed registration. Set the extended parameter to true to get more details

String API_GetDiscReasonText()
Return the disconnect reason of the last disconnected call.

String API_GetGlobalStatus()
Returns SIP stack state/details.

string API_GetLineStatusText(int line)
Get the line state as string.
This is rarely needed since you receive the status also by event notifications or you can use the API_GetStatus function instead.
(API_GetLineStatus is a similar function which returns the state as int/enum)

int API_GetAccountRegState(String domain, String proxy, String username, String sipusername, boolean pushnotify, boolean strict)
Query account register state. Useful if you are using multiple accounts.

Returns -1: no such account register ep, 0: working, 1: success, 2: failed, 3: unregistered.
Parameters domain, proxy, username, sipusername will be used to match the account.
Always set the pushnotify to false.
If the strict parameter is false, then will check the best matching account after the passed parameters.
If the strict parameter is true, then will enforce exact parameter match when checking after the endpoints (server domain/username/etc).

String API_GetAccountRegStateString(String domain, String proxy, String username, String sipusername, boolean pushnotify, boolean strict)
Query account register state. Useful if you are using multiple accounts.
Return status text (for example "Registered" if the account is registered to the server).
Parameters domain, proxy, username, sipusername will be used to match the account.
Always set the pushnotify to false.
If the strict parameter is false, then will check the best matching account after the passed parameters.
If the strict parameter is true, then will enforce exact parameter match when checking after the endpoints (server domain/username/etc).

String API_GetCallerID(int line)
Will return the remote party name

String API_GetIncomingDisplay(int line)
Get incoming caller id (might return two lines: caller id \n caller name)

String API_GetLastCallDetails()
Get details about the last finished call.

String API_GetProfileStatusText(String username)
Get the profile status text for the user.
Note: our profile status text can be set with the profilestatustext parameter.
This feature might be useful to implement a “My Profile” page in certain softphones.

String API_GetAddress()
Return the local SIP listener address (IP:port)

String API_GetMySIPURI(boolean all)
Will return the SIP URI on which the current endpoint can be reached such as username@server.com or username@localip:port.
If all is true, then it will return all possible URI’s separated by comma.

boolean API_SetSSLContext(SSLContext sc)
Set the SSLContext java object to be used for new SIPS/TLS connections.
You might use this if you have some special requirements regarding the TLS connections to configure an SSLContext as you wish and pass it to JVoIP.

boolean API_ShowLog()
Show a new window with logs.

void API_AddLog(String msg)
Add a message to JVoIP log.

String API_HTTPGet(String url)
Send a HTTP GET request. Check if return string begins with “ERROR”.

boolean API_HTTPPost(String url, String data)
Send a HTTP POST request.

String API_HTTPReq(String url, String data)
Send a HTTP POST or GET request. (If data is empty, then GET will be sent). Can be tunneled. Can block for up to 20 seconds.

boolean API_HTTPReqAsync(String url, String data)
Send a HTTP POST or GET request. (If data is empty, then GET will be sent). Can be tunneled. The result will be returned in notifications with “ANSWER” header.

boolean API_SaveFile(String filename, String content)
Will save the text file to local disk JVoIP working directory in encrypted format (use API_SaveFileRaw to save as-is)

String API_LoadFile(String filename)
Load file from local disk.

boolean API_SaveFileRemote(String filename, String content)
Save file to remote storage (preconfigured ftp or http server)

boolean API_LoadFileRemote(String filename)

Load file from remote storage. The download process is performed asynchronously. You need to call this function only once and then a few seconds later call the API_LoadFile function with the same file name. It will contain "ERROR: reason" text if the download failed.

String API_LoadFileRemoteSync(String filename)
Will download the specified file from remote storage synchronously (will block until done or fails).

String API_GetBlacklist()
Get the current blacklist.

boolean API_SetBlacklist(String str)
Set the whole blacklist (users/numbers/IP addresses separated by comma)

boolean API_AddToBlacklist(String str)
Add to blacklist (request from this user will be blocked)

String API_GetWhitelist()
Get the current whitelist.

boolean API_SetWhitelist(String str)
Set the whole whitelist (users/numbers separated by comma)

boolean API_AddToWhitelist(String str)
Add to whitelist (request from this user will be allowed)

boolean API_ClearCredentials()
Clear existing user account details.

boolean API_DelSettings (int level)
Delete settings and data.
Levels: 0: delete settings only if it was not already deleted, 1: delete settings, 2: delete all, 3: delete also library cache

int API_IsEncrypted ()
Returns whether the connection is encrypted. -1: unknown, 0=no,1=partially/weak yes,2=yes,3=always strong

String API_GetBindir()
Returns the application path (folder with the app binaries or app home folder)

String API_GetWorkdir()
Returns the application working directory (data folder, usually the "mwphonedata" subfolder or other location if no write access here).

String API_GetAltWorkdir()
Returns the application alternative working directory (such as folder on external SD card).

int API_ShouldReset()
Check if the sipstack should be restarted. Usually this is required only on local network change (such as IP change or changing from mobile data to wifi).
Possible return values: 0=no,1=not registered for a while,2=network changed

boolean API_ShouldResetBeforeCall()
This function might be called before calls and you should quickly restart the sipstack if returns true (the continue to make the call).

boolean API_RecFiles_Del()
Delete local recorded files.

boolean API_Log(String msg)
Send a log to JVoIP (to write it in its own logfile/console/etc). These are handled with loglevel 2 or higher.

boolean API_ReStart()
Restart the SIP stack.

boolean API_GetDeviceID()
Returns an unique device ID.

Notifications

Notifications means events received from the SIP library when something is happening/happened.
To be able to get events from JVoIP you will need to extend the SIPNotificationListener class and subscribe for the notification events with the API_SetNotificationListener function call. Then in your NotificatioListener subclass you will receive the notifications as SIPNotification objects or its subclasses.

Example:

```
class MySIPNotificationListener extends SIPNotificationListener
{
   public void onAll(SIPNotification e) {
      System.out.println("Notification received: " + e.toString());
   }
   public void onStatus (SIPNotification.Status e) {
      System.out.println("STATUS notification received: " + e. getStatusText());
   }
}

webphoneobj.API_SetNotificationListener(new MySIPNotificationListener());
```

See the javadoc for a more structured help about the notification objects and their fields.
Here is a simple working example code.

For maximum flexibility, the JVoIP SIP library implements also other ways to receive the SIP notifications as strings via API, JavaScript function or UDP/TCP socket. In case if somehow you are interested in these, see the details here.

The SIPNotification base class has a toString() function which can be used to convert the event to string. (The string format is better explained here)
Many notifications has a line field to be queried with the getline() function returning the SIP endpoint channel number as described here.
You don't necessarily have to handle all notification events in your SIPNotificationListener subclass. Override only the members you are interested at.

The easiest way to get started is to just log out all messages at first and from there the usage should be obvious.
Below we describe most notifications as strings and field names. See the javadoc for the actual SIPNotification class functions and/or use the auto-complete functionality from your IDE.

The following notification messages are defined:

Status

You will receive STATUS notifications when the SIP session state is changed (SIP session state machine changes) or periodically even if there was no any change.
Usually you will have to display the call state for the user, and when a call arrives you might have to display an accept/reject button.

Template string: STATUS,line,statustext,peername,localname,endpointtype
A typical status as string looks like this:
STATUS,line,statustext,peername,localname,endpointtype,peerdisplayname,[callid],online,registered,incall,mute,hold,encrypted,video,group, rtpsent,rtprec,rtploss,rtplosspercet,videohold,videosent,videorec,serverstats

Note: not all fields/functions are meaningful for all notifications. For example getRtpsent() should be queried only for endpoints in calls (status between STATUS_CALL_SETUP/ STATUS_CALL_CONNECT and STATUS_CALL_FINISHED) and it is meaningless for STATUS_REGISTER.

SIPNotification.Status methods:
getLine(): The line parameter can be -1 for global status or a positive value for the different lines.
Global status means the state of the "best" endpoint. For example if one line is disconnected and another is in Speaking, then the global state will be Speaking.
You can receive the same STATUS notification multiple times: for the particular endpoint, for the global status and repeatedly if the state remains the same.
You might decide to parse only general status messages (where the line is -1), messages for specific line (where line is a positive number) or both.
getStatus(): The global or endpoint state. One of the following constants: STATUS_UNKNOWN, STATUS_NOTREADY, STATUS_INITIALIZING, STATUS_READY, STATUS_OUTBAND, STATUS_REGISTER, STATUS_REGISTERED, STATUS_REGISTERFAILED, STATUS_UNREGISTER, STATUS_SUBSCRIBE, STATUS_MESSAGE, STATUS_CALL_SETUP, STATUS_CALL_ROUTED, STATUS_CALL_PROGRESS, STATUS_CALL_SESSIONPROGRESS, STATUS_CALL_RINGING, STATUS_CALL_CONNECT, STATUS_CALL_SPEAKING, STATUS_CALL_MIDCALL, STATUS_CALL_MUTE, STATUS_CALL_HOLD, STATUS_CALL_FINISHING, STATUS_CALL_FINISHED, STATUS_DELETABLE, STATUS_ERROR
getStatusText(): the status as string as described here
getPeer(): the other party username (if any)
getLocalname(): the local user name (or username).
getEndpointType(): check if it is a server or client endpoint. Will return one of the following constants:
DIRECTION_UNKNOWN
DIRECTION_OUT: for sessions initiated by JVoIP such as outbound calls where JVoIP acts as a client
DIRECTION_IN: for inbound sessions such as incoming calls when JVoIP acts as a server endpoint
getEndpointTypeText(): endpoint state as string
getPeerDisplayname(): the other party display name if any
getCallID(): SIP session id (SIP call-id)

getOnline(): network state. NETWORK_STATE_UNKNOWN, NETWORK_STATE_OFFLINE (no network or internet connection), NETWORK_STATE_ONLINE (connectivity detected)
getOnlineText(): online state as string
getRegistered(): registration state. REGISTER_STATE_UNKNOWN, REGISTER_STATE_NOTNEEDED, REGISTER_STATE_YES, REGISTER_STATE_WORKING_YES, REGISTER_STATE_WORKING_UNKNOWN, REGISTER_STATE_WORKING_NO, REGISTER_STATE_UNREGISTERED, REGISTER_STATE_FAILED
getRegisteredText(): registration state as string
getIncall(): phone/line is in call. CALL_STATE_UNKNOWN, CALL_STATE_NO, CALL_STATE_CONNECTING, CALL_STATE_CONNECTED
getIncallText(): call state as string
getMute(): is muted state. MUTE_STATE_NO, MUTE_STATE_UNKNOWN, MUTE_STATE_PLAY, MUTE_STATE_REC, MUTE_STATE_BOTH
getMuteText(): mute state as string
getHold(): is on hold state: HOLD_STATE_NO, HOLD_STATE_UNKNOWN, HOLD_STATE_SENDONLY, HOLD_STATE_RECVONLY, HOLD_STATE_BOTH
getHoldText(): hold state as string
getEncrypted(): encryption state: ENCRYPTION_STATE_UNKNOWN, ENCRYPTION_STATE_NO, ENCRYPTION_STATE_YES_WEAK, ENCRYPTION_STATE_YES, ENCRYPTION_STATE_YES_STRONG
getEncryptedText(): encryption state as string
getVideo(): video state: VIDEO_STATE_UNKNOWN, VIDEO_STATE_NO, VIDEO_STATE_INITIALIZED, VIDEO_STATE_OFFERED, VIDEO_STATE_RTPREADY, VIDEO_STATE_STARTED, VIDEO_STATE_STREAMING
getVideodText(): video state as string
getGroup(): group string for group chat and conference calls (members separated by |)
getRtpsent(): number of sent RTP packets (only if endpoint is in call)
getRtprec(): number of received RTP packets (only if endpoint is in call)
getRtploss(): number of lost RTP packets (only if endpoint is in call)
getRtplosspercent(): percent of the lost RTP packets (only if endpoint is in call)
getVideoHold(): video on hold state: HOLD_STATE_NO, HOLD_STATE_UNKNOWN, HOLD_STATE_SENDONLY, HOLD_STATE_RECVONLY, HOLD_STATE_BOTH
getVideoHoldText(): video hold state as string
getVideoRtpsent(): number of sent video RTP packets (only if endpoint is in video call)
getVideoRtprec(): number of received video RTP packets (only if endpoint is in video call)
getServerstats(): RTP statistics received from the server, if any (only if endpoint is in call)

See the SIPNotification.Status in the javadoc for the SIPNotification object details.

STATUS string examples:

The following status means that there is an incoming call ringing from 2222 on the first line:

STATUS,1,Ringing,2222,1111,**2**,Katie,[callid]

The following status means an outgoing call in progress to 2222 on the second line:

STATUS,2,Speaking,2222,1111,**1**,,[callid]

The following status means that the call was disconnected:

STATUS,2,Finished,2222,1111,1,,[callid]

Note:

- *To verify the connection/registration state you might use the below REGISTER notification instead or as described here.*
- *If the "stats" parameter is on (set to a value higher than 0) then you will receive periodic status messages during calls (might be useful if you are interested in RTP statistics during a call).*

Register

This notification is received for register state changes from registrar endpoints.
Registration means connection and authentication on your SIP server.
Template string: REGISTER,line,state,text,main,fcm,user,reason

SIPNotification.Register methods:

getLine(): channel number (should be always 0 for register)
getStatus(): registration state: STATUS_INPROGRESS, STATUS_SUCCESS, STATUS_FAILED, STATUS_UNREGISTERED
getText(): register state as string
getIsMain(): true for primary account, false for secondary registrations (if you are using multiple accounts)
getFcm(): not used (ignore)
getUser(): local username (useful if you are using multiple accounts)
getReason(): failure reason text if any

See the SIPNotification.Register in the javadoc and the How to register FAQ point for more details.

Presence

This notification is received for presence changes (peers online state).
Template string: PRESENCE,peername,state,details,displayname,email

SIPNotification.Presence methods:

getPeer(): username of the peer
getStatus(): presence state. One of the following constants: PRESENCE_UNKNOWN, PRESENCE_OFFLINE, PRESENCE_DND, PRESENCE_AWAY, PRESENCE_BUSY, PRESENCE_ONLINE, PRESENCE_PENDING
getStatusText(): presence state as string
getDetails(): presence details if any
getPeerDisplayname(): peer full name if received (it can be empty)
getEmail(): peer email address if received (it can be empty)

See the SIPNotification.Presence in the javadoc for the SIPNotification object details.
More details about presence handling can be found here.

Notes for the state and detail strings:
The state and the details strings are usually the followings:
CallMe,Available,Open,Pending,Other,CallForward,CallSetup,Speaking,Busy,Idle,DoNotDisturb,DND,Unknown,Away,Offline,Closed,Close,Unreacheable, Unregistered,Invisible,Exists,NotExists,Unknown,Not Set or as reported by the peer
One of these fields might be empty in some circumstances and might not be a string in the above list (especially the details).
The details field will provide a more exact description (for example "Unreachable") while the state field will provide a more exact one (for example "Close"). For this reason if you have a presence control to be changed and you are looking for the strings instead of using the getstatus function, then check the details string first and if you can't recognize its content, then check the state string. For displaying the state as text, you should display the details field (and display the state field only if the details string is empty).

BLF

This notification is received for incoming BLF messages.
Template string: BLF,peerusername,direction,state,callid
Described at the BLF section above.

DTMF

Incoming DTMF notification.
Msg is a string parameter representing the incoming DTMF digit. Usually one of the followings: 0123456789*#ABCDF
Supported receive dtmf methods are SIP INFO in SIP signaling and NTE (RFC 2833 in RTP).
Template string: DTMF,line,msg
Example: DTMF,1,7 (dtmf digit"7" received on first line)

SIPNotification.DTMF members:

getLine(): returns the channel number (the endpoint which received the DTMF message)
getMsg(): returns the DTMF message (one or more DTMF digits)

See the SIPNotification.DTMF in the javadoc for the SIPNotification object details.

INFO

Notifications about incoming/outgoing INFO or DTMF messages.

Note: for incoming DTMF you should watch for the above described DTMF notification instead of this! DTMF notification will be triggered even if the message was received with SIP INFO.

Template string: INFO,type,line,peername,text

SIPNotification.INFO members:
getType(): will return one of the following constants:

- INFO_UNKNOWN: this should never be emitted
- INFO_ERROR: Outgoing DTMF (API_Dtmf) or INFO (API_Info) message failed
- INFO_WARNING: Outgoing DTMF failover from SIP INFO to RFC 2833 or In-Band on answer timeout or error response
- INFO_OK: Outgoing DTMF (API_Dtmf) or INFO (API_Info) was sent successfully
- INFO_REC: Incoming INFO message received. Note that if the incoming message is a dtmf digit, then a DTMF notification will be triggered instead of INFO

getTypeText(): type as string
getLine(): Phone line
getPeer(): Other party username
getText(): INFO message body when type is REC or any additional info if type us ERROR, WARNING or OK.

See the SIPNotification.INFO in the javadoc for the SIPNotification object details.

Examples:

INFO,ERROR,1,1111,timeout (on API_Dtmf or API_Info message send failed to 1111 on line 1)
INFO,WARNING,1,1111,failback to rfc2833 (on API_Dtmf failback from INFO to rfc2833 or in-band)
INFO,OK,1,1111 (on API_Dtmf or API_Info message sent successfully to 1111 on line 1)
INFO,REC,1,1111,hello (on SIP INFO message received from 1111 which is not DTMF)

Note:
When sending DTMF in RTP (inband or rfc2833) then it is not possible to get feedback whether the message was actually delivered. In this case you will receive an INFO,OK when the message send was initiated successfully. Otherwise if SIP INFO is used, then you will receive INFO,OK only on 2xx answer from the server or the other peer.

USSD

This notification is received about incoming USSD messages or report about success/failure about the outgoing USSD messages sent by API_SendUSSD.
Template string: USSD,line,status,text

SIPNotification.USSD members:

getLine(): the session line
getStatus():
- USSD_UNKNOWN: should never be emitted
- USSD_FAILED: send failed
- USSD_SENT: send succeeded
- USSD_RECEIVED: received

getStatusText(): the status as string
getText(): text is the received USDD string (if status is 2) otherwise it might contain an error (if status is 0)

See the SIPNotification.USSD in the javadoc for the SIPNotification object details.
More details about USSD handling can be found here.

Chat

This notification is received for incoming chat messages.
Template string: CHAT,line,peername,text

SIPNotification.Chat members:

getLine(): used phone line
getPeer(): username of the peer
getText(): the message body as-is
getMsg(): chat text
getGroup(): group name (only for multi-user/group messages, otherwise will return empty string)
getMultiparCurrent(): multipart current fragment (only for 3gpp multipart messages)
getMultiparTotal(): multipart total fragments (only for 3gpp multipart messages)

See the SIPNotification.Chat in the javadoc for the SIPNotification object details.

ChatReport

Chat transmission status report so you can process if outgoing message sent with the API_SendChat was delivered successfully or failed.
Template string: CHATREPORT,line,peername,status,statustext,group,md5,id

SIPNotification.ChatReport members:

getLine(): used phone line
getPeer(): username of the peer
getStatus(): one of the following constants:
- STATUS_UNKNOWN: should never be emitted
- STATUS_INIT: means chat send initialized
- STATUS_SENDING: means sending in progress
- STATUS_SENT: means successfully sent
- STATUS_FAILED: means failed
- STATUS_QUEUED: means queued for later send

getStatusText(): status text if any (such as delivery error reason if status is 3)
getGroup(): optional parameter for group chat (member names separated by |)
getMD5(): the MD5 checksum of the message text
getID(): message ID (user supplied msgid, 3gpp message reference or generated id)

See the SIPNotification.ChatReport in the javadoc for the SIPNotification object details.

Note:

This notification is sent also for SMS messages.

If the offline messaging is not disabled (offlinechat parameter is not set to 0) then JVoIP can retry later (on next start and/or when any SIP request is received from the target user) reporting status 4 instead of 3 on failed delivery. The MD5 checksum in this case will be calculated from concatenated pending message text and not for the last message.

A STATUS_QUEUED is reported also if your SIP server responds with 202 (Accepted) instead of 200 (OK). This (the 202 response code) should indicate that the request has been accepted but the processing has not been completed, however some servers (such as Asterisk) might always (incorrectly) answer with 202 for the MESSAGE requests instead of 200. You might set the chatacceptasok parameter to 1 to report status STATUS_SENT for these instead of status STATUS_QUEUED.

ChatComposing

Chat "is composing" notifications are usually emitted when the other party starts or stops typing.
Template string: CHATCOMPOSING,line,peername,status

SIPNotification.ChatComposing members:

getLine(): used phone line
getPeer(): username of the peer
getStatus(): STATUS_TYPING means other party is typing, STATUS_IDLE means other party is idle or stopped typing
getStatusText(): status as string

See the SIPNotification.ChatComposing in the javadoc for the SIPNotification object details.

CDR

After each call, you will receive a CDR (call detail record) with the details about the call.
Template string: CDR,line, peername,caller,called,peeraddress,connecttime,duration,discparty,reasontext

SIPNotification.CDR members:

getLine(): used phone line
getPeer(): other party username, phone number or SIP URI
getCaller(): the caller party name (our username in case when we are initiated the call, otherwise the remote username, displayname, phone number or URI)
getCalled(): called party name (our username in case when we are receiving the call, otherwise the remote username, phone number or URI)
getPeerAddress(): other endpoint address (usually the VoIP server IP or domain name)
getConnectTime(): milliseconds elapsed between call initiation and call connect (or until reject/hangup if the call was not connected)
getDuration(): milliseconds elapsed between call connect and hangup (0 for not connected calls. Divide by 1000 to obtain seconds.)
getDiscParty(): the party which was initiated the disconnect:
- DISCBY_NOTSET: not set
- DISCBY_LOCAL: local JVoIP
- DISCBY_REMOTE: peer
- DISCBY_UNKNOWN: undefined/timeout

getDiscReason(): a text about the reason of the call disconnect (SIP disconnect code, CANCEL, BYE or some other error text)

See the SIPNotification.CDR in the javadoc for the SIPNotification object details.

Start

This message is sent immediately after startup (so from here you can also know that the SIP engine was started successfully).
Template string: START,what

SIPNotification.Start members:

getWhat(): START_API or START_SIP
getWhatText(): the what value as string:
"api" -api is ready to use
"sip" –sipstack was started

See the SIPNotification.Start in the javadoc for the SIPNotification object details.

Stop

This message is sent when the SIP stack is terminated/destroyed.
Template string: STOP,api

SIPNotification.Stop members:

getWhat(): STOP_API or STOP_SIP

getWhatText(): the what value as string:

"api" -api is not available anymore

"sip" –sipstack was stopped (might not be emitted)

See the SIPNotification.Stop in the javadoc for the SIPNotification object details.

ShouldReset

You should restart JVoIP (or re-init the library) when you receive this notification.
This is usually sent when network was changed (connection type, IP address), thus it might be best to reinitialize the whole SIP stack to recalculate the optimized environment variables and perform the auto network discovery process again (such as STUN lookup, but there are many others).
The message is not sent while in call.
Template string: SHOULDRESET,reason text

SIPNotification.ShouldReset members:

getReason(): reason text

See the SIPNotification.ShouldReset in the javadoc for the SIPNotification object details.

Line

This message is sent when the active line is changed (the line parameter is the current active line).
Template string: LINE,line

SIPNotification.Line members:

getLine(): the new active line number

See the SIPNotification.Line in the javadoc for the SIPNotification object details.

Popup

Should be displayed for the users in some way (hint/toast).
Template string: POPUP,txt

SIPNotification.Popup members:

getText(): the message string

See the SIPNotification.Popup in the javadoc for the SIPNotification object details.

Event

Important events which should be displayed for the user.
Template string: EVENT,TYPE,txt
Example: EVENT,EVENT,any text

SIPNotification.Event members:

getType(): the event type: TYPE_EVENT, TYPE_WARNING, TYPE_ERROR, TYPE_CRITICAL, TYPE_UNKNOWN

getTypeText(): the type as string: EVENT, WARNING, ERROR

getText(): the message string

See the SIPNotification.Event in the javadoc for the SIPNotification object details.

Note: These messages will not be received if you set the "events" parameter below 2 (default is 2) or the loglevel parameter below 1.

Log

Detailed logs (may include also the SIP signaling).
Template string: LOG,TYPE,txt

SIPNotification.Log members:

getType(): the log type: TYPE_EVENT, TYPE_WARNING, TYPE_ERROR, TYPE_CRITICAL, TYPE_UNKNOWN
getTypeText(): the type as string: EVENT, WARNING, ERROR, CRITICAL, RTP
getText(): the log string

See the SIPNotification.Log in the javadoc for the SIPNotification object details.

Note: These messages will be received only if you set the events parameter to 3 and also depends on the loglevel.

With RTP you might receive the media stream details at the end of the calls.
Example: RTP: sent 15695 lasts: 0 (p2p: 0 sdp: 0 rrp: 15695 tnl: 15701), rec: 17281 lastr: 0, loss: 201 1%, vsent: 0/0, vrecv: 0/0, cpu: 0.0%, cpurel: 0.0% (0.0), srvsent: 10326 srvrec: 10302 srvloss: 10 0%

Group

Sent for conference calls. You might update the displayed peer name with the peers strings received here.
Template string: GROUP,line,peers

SIPNotification.Group members:

getLine(): endpoint channel number
getPeers(): list of members separated by |. For example: Kate | John | Linda

See the SIPNotification.Group in the javadoc for the SIPNotification object details.

Vrec

Voice upload status (for voice recording / call recording).
Template string: VREC,line,stage,type,path,reason,source

SIPNotification.Vrec members:

getLine(): channel number *(note: with stage 3 and 4 it will always report -1 or -2 means default/not specified)*
getStatus(): call recording state:
- STAGE_DISABLED: no such state should be reported
- STAGE_INIT: call record begin
- STAGE_SAVING: save begin
- STAGE_SAVED: save success
- STAGE_FAILED: save fail
- STAGE_UNKNWON: no such state should be reported

getStatusText(): state as string
getType(): upload method:
- TYPE_UNKNOWN: unknown
- TYPE_FILE: local file
- TYPE_FTP: ftp
- TYPE_HTTP: http
- TYPE_SERVER: server
- TYPE_SIPREC: siprec

getTypeText(): the type as string
getPath(): upload URL or file path *(note: if stage is 1 then type and path is not reported yet)*
getReason(): failure reason *(if stage is 4)*
getSource(): who initiated the recording: SOURCE_USER, SOURCE_SIPREC, SOURCE_AUTO, SOURCE_UNKNOWN
getSourceText(): the source as text (USER, SIPREC or AUTO)

See the SIPNotification.Vrec in the javadoc for the SIPNotification object details and the voice recording FAQ for more details.

PlayReady

Audio streaming finished (initiated by API_PlaySound to remote peer).
Template string: PLAYREADY,line,callid

SIPNotification.PlayReady members:

getLine(): endpoint channel number
getCallID(): SIP Call-ID

See the SIPNotification.PlayReady in the javadoc for the SIPNotification object details.

Note: this notification is not sent with call disconnect (call disconnect will always terminate the audio streaming).

SIP

Notifications for received/sent SIP signaling messages.
Can be enabled by the sipmsgnotifications parameter (disabled/not sent by default)
Template string: SIP,direction,address,message

SIPNotification.SIP members:

getDirection(): DIRECTION_UNKNOWN, DIRECTION_OUT, DIRECTION_IN
getDirectionText(): "in" (for outgoing messages sent by JVoIP) or "out" (for incoming messages received by JVoIP)
getAddress(): peer IP:port
getMessage(): the whole SIP signaling message text

See the SIPNotification.SIP in the javadoc for the SIPNotification object details.

Note: if the message already belongs to a session, then you can get the endpoint line by extracting the Call-ID value and passing it to the API_CallIDToLine() function.

Block

This notification is emitted when JVoIP blocks a signaling message only if the sendblocknotifications parameter is set to 1.
Template string: BLOCK,type,message

SIPNotification.Block members:

getType(): TYPE_UNKNOWN, TYPE_BLACKLIST, TYPE_WHITELIST
getTypeText(): the type as string
getMessage(): the SIP message being blocked / ignored

See the SIPNotification.Block in the javadoc for the SIPNotification object details.

VAD

Voice Activity Detection.
This notification can be used to detect speaking/silence or to display a visual voice activity indicator.

It is sent in around every 3000 milliseconds (3 seconds) by default (configurable with the vadstat_ival parameter in milliseconds) if you set the "vadstat" parameter to 3 or 4 it can be requested by API_VAD. Also make sure that the "vad" parameter is set to at least "2".
VAD notifications with the exact same parameters might not be resent.

Template string: VAD,parameters

String format:
VAD,local_vad: ON local_avg: 0 local_max: 0 local_speaking: no remote_vad: ON remote_avg: 0 remote_max: 0 remote_speaking: no

If the vadbyline parameter is set to 1 then individual lines will report in the following format:
VAD,line: X remote_vad: ON remote_avg: 0 remote_max: 0 remote_speaking: no

SIPNotification.VAD members:
getLine(): endpoint channel number
getParameters(): returns all parameters

getLocalValid(): whether VAD is measured for microphone: ON or OFF
getLocalAvg(): average signal level from microphone
getLocalMax(): maximum signal level from microphone
getLocalSpeaking(): local user speak detected: yes or no

getRemoteValid(): whether VAD is measured from peer to speaker out: ON or OFF
getRemoteAvg(): average signal level from peer to speaker out
getRemoteMax(): maximum signal level from peer to speaker out
getRemoteSpeaking(): peer user speak detected: yes or no

See the SIPNotification.VAD in the javadoc for the SIPNotification object details.

More details about VAD handling can be found here.

RTPE

Reports about RTP extra header changes.
This is sent only if there RTP header extensions are received and the rtpextraheadernotify parameter is set to 1.
Template string: RTPE,line,profile,extension
Example: RTPE,1,1,98;76543

SIPNotification.RTPE members:

getLine(): the session endpoint line number
getProfile(): identifier or parameter extracted from the first two byte of the extension header
getExtension(): 32 bit words as int converted to string. If there are more then one then separated by semicolon (;)

See the SIPNotification.RTPE in the javadoc for the SIPNotification object details.
See the RTP header extension FAQ point for more details about RTPE.

RTPStat

Media quality reports.
Can be enabled by the rtpstat parameter.
The parameters are calculated since the last RTPSTAT notification (for the past x seconds).
Template string: RTPSTAT,quality,sent,rec,issues,loss

SIPNotification.RTPStat members:

- **getQuality()**: quality points between 0 and 5. The calculations considers many factors such as RTP issues, RTCP reports, codec problems, packet loss, packet delay, jitter and processing delay.
  The following values are defined:
  - QUALITY_UNKNOWN: unrecognizable (-1)
  - QUALITY_NO: no audio or non-recognizable voice (0)
  - QUALITY_LOWEST: very bad quality (1)
  - QUALITY_LOW: bad quality (2)
  - QUALITY_MEDIUM: medium quality (3)
  - QUALITY_HIGH: good quality (4)
  - QUALITY_HIGHEST: excellent quality (5)
- **getQualityText()**: the quality as string
- **getSent()**: RTP packets sent
- **getRec()**: RTP packets received and played
- **getIssues()**: number of issues (any issues are counted, such as sequence number mismatch or packet drop)
- **getLoss()**: lost packets

See the SIPNotification.RTPStat in the javadoc for the SIPNotification object details.

More details here.

MWI

Voicemail notifications.
Messages are received on new voicemail notifications if you have enabled voicemail and there are pending new messages.
This notification might be set by the SIP server without asking or depending on the voicemail parameter or requested by API_CheckVoicemail.
Template string: MWI,hasvoicemail,voicemailnumber,to,count,message
Example string: MWI,yes,5001,1111,3,3/0

SIPNotification.MWI methods:

- **getHasMessage()**: “yes” or “no” (Messages-Waiting indicator, whether if there are message(s) pending)
- **getVMNumber()**: the voicemail access number (as configured or as received in Message-Account. see voicemailnum)
- **getTo()**: username from the SIP To header (Message-Account; usually the local account username, useful if you have multiple accounts)
- **getCount()**: number of new pending messages (Messages-Waiting)
- **getMessage()**: message text if sent by the server (Voice-Message)

See the SIPNotification.MWI in the javadoc for the SIPNotification object details.

SRS

Session recording server notification.
Template string: SRS,line,session_id,sip_callid,sipsessionid1,userid1,aor1,name1,sipsessionid2,userid2,aor2,name2,codec
Details here.

VIDEO

Video start/stop notification.
Template string:
VIDEO,startstop,type,line,reason,ip,port,codec,payload,quality,bw,max_bw,fps,max_fps,width,height,profilelevelid,profile,pixelfmt,level,pm,sprop,srtp_alg,srtp_key,srtp_remotekey,device,fmtp
Described in the separate video documentation. More details here.

Other notifications

Some other rarely required/used notifications are described below:
Format: messageheader, messagetext.

"**CREDIT**" messages are received with the user balance status if the server is sending such messages.
Example: "CREDIT,Credit: 2 USD"
See the SIPNotification.Credit in the javadoc for the SIPNotification object details.

"**RATING**" messages are received on call setup with the current call cost (tariff) or maximum call duration if the server is sending such messages.
Example: "RATING,15 cents/min"
See the SIPNotification.Rating in the javadoc for the SIPNotification object details.

"**NEWUSER**" new contact request
Example: NEWUSER,usrname,displayname
See the SIPNotification.NewContact in the javadoc for the SIPNotification object details.

"**SERVERCONTACTS**" contact found at local VoIP server (only by supported servers)
Example: SERVERCONTACTS,details
See the SIPNotification.ServerContacts in the javadoc for the SIPNotification object details.

"**ANSWER**" answer for previous request (usually http requests)
Example: ANSWER,RESULT:the_answer, REQUEST:the_request,EOFANSWER
See the SIPNotification.Answer in the javadoc for the SIPNotification object details.

Example session

Here is how the notifications looks like as strings in a typical session (start, register, make a call, hangup, terminate):

```
START,api
START,sip
STATUS,-1,Initialized
STATUS,-1,Register...
EVENT,EVENT,Connecting...
STATUS,-1,Registering...
STATUS,0,Registering,istvantest2,istvantest2,1,voip.mizu-voip.com,[2e112329450424958431 1k49333rmwp],,1,4,0,0,0,0,0
CREDIT,Credit: 4.2 USD
STATUS,-1,Registered.
EVENT,EVENT,Authenticated successfully. [ep: 0 2e112329450424958431 1k49333rmwp Registering 12538]
STATUS,0,Registered,istvantest2,istvantest2,1,voip.mizu-voip.com,[2e112329450424958431 1k49333rmwp],,1,2,0,0,0,0,0
STATUS,-1,Starting call to testivr3
EVENT,EVENT,call [wpmain]
STATUS,-1,Call
STATUS,-1,Call Initiated
STATUS,1,CallSetup,testivr3,istvantest2,1,testivr3,0,0,0,100,,[6e4351661777543861707k49334rmwp],1,2,1,0,0,0,0
STATUS,1,Routed,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,1,0,0,0,0
STATUS,1,InProgress,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,1,0,0,0,0
STATUS,-1,Calling...
STATUS,1,Ringing,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,1,0,0,0,0
STATUS,-1,Ringing...
STATUS,1,CallConnect,testivr3,istvantest2,1,testivr3,0,0,0,100,,[6e4351661777543861707k49334rmwp],1,2,2,0,0,0,0
STATUS,1,Speaking,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,2,0,0,0,0
STATUS,-1,Speaking (0 sec)
STATUS,-1,Hangup
STATUS,-1,Speaking (2 sec)
EVENT,EVENT,hangup [wpmain]
STATUS,1,CallDisconnect,testivr3,istvantest2,1,testivr3,98,98,0,100,,[6e4351661777543861707k49334rmwp],1,2,0,0,0,0,0
STATUS,-1,Call Finished
CDR,1,testivr3,istvantest2,testivr3,voip.mizu-voip.com,1499,1969,1,User Hung Up (exD),[6e4351661777543861707k49334rmwp]
```

STATUS,1,Finishing,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,0,0,0,0,0
EVENT,EVENT,Call duration: 2 sec [ep: 1 6e4351661777543861707k49334rmwp Finishing 12538]
STATUS,1,Finished,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,0,0,0,0,0
STATUS,-1,Registered.
STOP,api

Parameters

"Parameters" here means JVoIP settings (configurations / options).
Parameters can be specified in the following ways:

- command line (when used as a standalone executable. Example: JVoIP.jar serveraddress=1.2.3.4 username=xxx password=xxx loglevel=5)
- config file: wpcfg.ini file in ini file format
- using the API_SetParameter function (this is the most commonly used method if you use JVoIP as a java library)
- SIP signaling (sent from server) with the x-mparam header (or x-mparamp if need to persist). Example: X-MParam: loglevel=5;aec=0
- environment variables (prefix parameter names with "wp_" in this case. For example: wp_username)
- if used from a webpage (applet):
  applet parameters (by using the applet tag and set the parameters like: <param name = "parameter_name" value = "parameter_value">
  webpage URL (the webhone will simply look at the embedding document/window url at startup. Prefix all parameter with "wp_")
  cookies (prefix cookie keys with "wp_". For example wp_username)
  skin config file: config.js (if you are using our skin templates)

Any of these methods can be used or they can be even mixed.
All parameters can be passed as strings and will be converted to the proper type internally by the VoIP SDK.
Parameters are usually set at startup (before API_Start or loaded from config file or command line), but most of the parameters can be changed also at runtime (for example before main actions / API calls or between two calls if the second calls needs some other settings).

For a basic usage you will have to set only your VoIP server ip or domain name ("serveraddress" parameter)
The rest of the parameters are optional and should be changed only if you have a good reason for it as all parameters has a best/optimal/auto-guess default value.

Note:

- Parameters can be also encrypted.
- Some parameters (especially basic networking related, such as signalingport) can't be changed at runtime (Change attempts will not cause fatal errors but might be ignored or not loaded. In case if you need to change basic settings then it is recommended to stop the JVoIP instance first and (re)start it with the new parameters set at start-up time).
- Once a parameter is set, it might be cached by JVoIP and used even if you remove it later. To prevent this, set the parameter to "DEF" or "NULL". So instead of just deleting, set its value to "DEF" or "NULL". "DEF" means that it will use the parameter default value if any. "NULL" means empty for strings, otherwise the parameter default value. Example: transport=DEF. Alternatively you might use the API_DelSettings function or set the resetsettings parameter to true.

Main Parameters

The parameters can be used to control the most important settings and behavior like server domain, SIP authentication parameters, called party number and whether a call have to be started immediately as JVoIP starts or you let the user to enter these parameters manually.

serveraddress

(string)
The domain name or IP (IPv4 or IPv6 address of your SIP server. By default it uses the standard SIP port (5060). If you need to connect to other port, you can append the port after the address separated by colon.
Examples:

mydomain.com" -this will use the default SIP port: 5060
sip.mydomain.com:5062
10.20.30.40:5065
[ipv6]:port

Default value is empty.

If the server address is not set with this serveraddress parameter, then alternatively it can be set with the API_Register or API_Call functions (if domain part is passed).
If the serveraddress was not set, then you (or the endusers) will be able to make calls only by using full SIP URI and it is more difficult to accept incoming calls (but still perfectly fine: see the P2P FAQ point)

If set, then any username/phone number can be called what is accepted by your server (it is enough to pass the username part in this case, no need to specify the full SIP URI for calls). As the SIP server you can use any softswitch, IP-PBX, SIP proxy server or SIP gateway/SBC (with or without registrar).
It might be possible that your service requires also (outbound) SIP proxy configuration, which can be set with the proxyaddress parameter. Always specify the proxyaddress If you are using a logical non-resolvable SIP domain/realm as the serveraddress (for example with multi-tenant servers where the tenant name is not a resolvable domain name).

username

(string)
This is the SIP username (A number/Caller-ID for the outgoing calls). The JVoIP SIP endpoint will authenticate with this username on your SIP server.
Default value is empty.

Note:
You should set only the username part of your SIP account here, not the full SIP URI. For example if your account looks like abc@xyz.com, then you should set xyz.com as the serveraddress parameter and abc as the username parameter.
If you need a different name for SIP user name (extension ID) and authentication username (authorization name) then use also the sipusername parameter.
You might also specify the display name. More details here.

password

(string)
SIP authentication password.
If your server doesn't require digest authentication or if you wish to make peer to peer calls then this parameter can be omitted.
This parameter can be set also in encrypted format or you can use the md5 parameter instead of the password. More details about the parameters encryption can be found here.
Default value is empty.

register

(int)
With this parameter you can set whether the SIP UA should register (connect) to the sip server at startup.

Possible values:

- 0: no
- 1: auto guess (yes if serveraddress/username/password is set, otherwise no)
- 2: yes

Default value is 1.

You might also/instead use the API_Register function call to register (Set the register parameter to 0 and call API_Register later when you need it).
JVoIP will also reregister automatically based on the registerinterval parameter (there is no need to periodically call the API_Register).

Other Parameters

These parameters are more rarely used or should be used only if you have at least a minimal technical knowledge (VoIP and/or Java). You should modify only those parameters which you fully understand otherwise better if you leave all with the default values (the default values are already optimized for production).

autocall

(boolean)
If set to true then the VoIP SDK will immediately starts the call with the given parameters (for example with your page load). The serveraddress, username, password, callto must be also set for this to work.
Default value is false.

Usually when this parameter is true, then the "compact" parameter is also set true. Usually when this parameter is false, then the "compact" parameter is also set false.

callto

(string)
Can be any phone number/username that can be accepted by your server or a SIP URI. When "autocall" or "compact" is true, then this parameter should be filled properly. Otherwise it can be empty or omitted (the user will enter the number to call)
Default value is empty.

codebase

(string)
Applicable only if you use it as applet from browsers.
This optional attribute specifies the base URL of JVoIP--the directory that contains JVoIP code. If this attribute is not specified, then the document's URL is used (your html page URL).
Use it if you deploy JVoIP.jar in a different directory on your webserver (not the same directory as your webpage).
For the toolkit deployment use “JAVA_CODEBASE” instead of “codebase”.
Default is ‘.’ which means the same directory (the html document directory)

use_rport

(int)
Check rport in SIP signaling (requested and received from the SIP server by the VIA header)
0=don’t ask (rtport request will not be added to the VIA header)
1=use only for symmetric NAT (only when it is sure that the public address will be correct)
2=always (always request and use the returned value except if already on public ip)
3=request even on public IP (meaningless in most cases)
9=request with the signaling, but don’t use the returned value (good If you want to keep the local IP and for peer to peer calls)
Default is 1.
Change to 0, 2 or 9 only if you have NAT issues (depending on your server type and settings).
You might adjust also the use_fast_stun parameter if you change the use_rport.
More details can be found here.

upnpnat

(int)
Nat traversal via UPnP supported routers (IGD)
0: disable
1: enable
Default is 1

use_fast_ice

(int)
Fast ICE negotiations (for p2p rtp routing):
-1: suggest server side media routing (X-P2PM: 1)
0: no (set to 0 only if your server needs to always route the media)
1: auto
2: yes
3: always (not recommended)
Default is 1

Note:
If set to 1 or 2 then the stun should not be disabled.
If enabled, then the media might be negotiated directly between the endpoint (without the use of server suggestions in the SDP) and might be routed directly encrypted.

use_fast_stun

(int)
Fast stun request on startup.
-1=force private address (if the client has both private and public IP, than the private IP will be sent in the signaling)
0=no
1=use only for stable symmetric NAT
2=use only if both tests match even if not symmetric (recommended)
3=use for symmetric NAT even if only one match
4=always
5=use even on public IP

Default is 2.

Change if you have NAT issues (depending on your server type and settings). You might adjust also the use_rport parameter if use_fast_stun is changed.
If your router or NAT device often changes the external port binding then you might set it to 0 so JVoIP will use the private address to avoid any confusion on the SIP server (in this case you should also set the use_rport parameter to 9.
If your SIP server doesn’t have any NAT support, then you should set it to 3 or 4 and you might set also the use_rport parameter to 2.
More details can be found here.

udpconnect

(int)
Specify whether the UDP have to be connected before sending on the socket. (Some IP-PBX might require udp connect and in this way the VoIP SDK can always detect its local address correctly. However this should not be used whit multiple servers or separate domain and outbound proxy)
0=no
1=on init
2=on first send (not recommended. can block)
3=on both or any (not recommended)
Default value: 0

keepalivetype

(int)
NAT keep-alive packet type.
0=no keep-alive
1=space + CRLF (_\r\n) (very efficient because low bandwidth and low server usage)
2=NOTIFY (standard method)
3=CRLF (\r\n) (very efficient because low bandwidth and low server usage. Not recommended)
4=CRLFCRLF (\r\n\r\n) (very efficient because low bandwidth and low server usage. After RFC draft)

Default is 4.

The keepalivetype 4 also conforms to RFC 5626 (the ping request is \r\n\r\n and the pong answer is \r\n).

keepaliveival

(int)
NAT keep-alive packet send interval in milliseconds.
Keep-alive is a mechanism to keep the NAT open between register resends (so the server can initiate a new incoming transaction on the same stream with no issues, such as incoming call).

Possible values:
- -1: auto (will default to around 28000 -28 sec- on UDP and 600000 – 10 min- on TCP)
- 0: disabled
- other: interval in milliseconds (must be above 3000, otherwise treated as seconds)

Default value is -1.

reconnectonnokeepalive

(int)
Specify if to re-register/reconnect on no answer for keep-alive.
0=no
1=yes (will reregister after around keepaliveival + 22 seconds)
2=yes, more aggressively (will reregister after around keepaliveival + 8 seconds)
Default is 1.

Usually SIP endpoint will detect connection loss only when there is no answer for REGISTER.
With this feature, JVoIP is capable to detect connection loss earlier.
When there is no answer for a keep-alive request, it will repeat it a few times (for 7-25 seconds) and if there is still no answer, then it will send a REGISTER request. If there is no answer also for the REGISTER, then it will try to re-register/reconnect/completely restart the SIP stack.
This feature will work only if your SIP server has support for keep-alive packets and no answers will be ignored if no any keep-alive answer have been received before.

registerinterval

(int)
Registration interval in **seconds** (used by the re-registration expires timer / register timeout interval).
Default value is -1 (auto guess optimal value)

Valid range is -1 (auto) or between 10 (10 seconds) and 86400 (one day).

Notes:
- *Many servers will not accept values above 3600 or below 60 (in this case the SIP stack will try to adjust it based on server negotiation)*
- *The actual resend of the REGISTER messages might be done at a shorter interval to cover any potential packet loss and network/server delays (usually in less than half interval, configurable also with the actualregisterexpires_val or actualregisterexpires_div settings)*
- *When set to auto guess (-1), it will calculate an optimal interval depending on the circumstances such as network type, server load, transport method. It will result to around 90-180 under normal conditions on UDP.*

- *If your softswitch supports keep-alive messages (to prevent NAT binding timeouts), then you might set to a longer interval (~3600 sec) to prevent high CPU usage on your registrar server, especially if you have many hundreds of SIP UA running at the same time. If your server doesn't support keep-alive, then you might set this to a lower value (between 30 and 90 sec. 60 sec is a good choice for most NAT devices and routers). Note that usually this is not necessary because server side support is not needed to keep the NAT bindings (so the keep-alive packets will keep the NAT port binding alive, even if your server doesn't respond to them).*
  See the keepalivetype, keepaliveival and reconnectonnokeepalive parameters for more details.
- *If the register expire interval is not accepted by the server, then the SIP stack is capable to automatically negotiate a new value as directed by the server Min-Expire header or auto-guess.*
- *If the SIP server doesn't answer for the REGISTER (after a few retry) then JVoIP will consider as connection lost and will perform additional actions to reconnect (a few more re-register, then a new register session and then a complete SIP stack restart –if there is no ongoing calls). If it can't reconnect, then it will continue to try normal reconnect periodically.*

needunregister

(boolean)
Set to false to prevent unregister messages to be sent by JVoIP (for example to prevent unregister on web page reload).
Default is true

unregall

(int)
Set to 1 to unregister all endpoints for the users, not only the current one.
Will send Contact: * with the unregister requests (REGISTER with Expires: 0).
Default is 0.

unregonstart

(int)
Set to 1 to unregister at startup (before to register)
Default is 0.

setinstanceid

(int)
Set to 1 to enable RFC 5626 behavior (connection flows), sending reg-id, +sip.instance and the ob parameter with the SIP Contact header.
Default is 0 (disabled).

contactaddressfailback

(int)
Specify if to try to reregister with another (better) local address if register fail or if previous (first) register went with a private address (and the server is on public IP).
Possible values:

- -1: auto (same as 2 if server is on public IP and stun/rport are not disabled and server is not known NAT friendly; otherwise same as 1)
- 0: never
- 1: if failed (will try to reregister with other local address on register failure)
- 2: always (also try to reregister with public address when possible which helps for servers with poor NAT support)

Default: -1

Note:
With the unbindbeforereregister parameter you can also specify if to unbind (unregister) the previous (private) address before registering the new one.
This might be forced on servers with no or poor call fork support, otherwise most servers should be able to just replace the previous address without the need to unregister.
Possible values: -1: auto, 0: never, 1: always (default).

extraregisteraccounts

(string)
Use this setting for multi-account registration.
You can specify one or multiple (up to 99) extra SIP accounts as SIP URI's or in the following format (parameters separated by comma, accounts separated by semicolon ;):
IP,usr,pwd,t,proxy,realm,authuser,displayname,trasnport;IP2,usr2,pwd2;IP3,usr3,pwd3, , ,realm3;IP4,usr4,pwd4…
where:

- IP: is the SIP server address (domain or IP:port)
- usr: is the SIP username
- pwd: is the SIP password
- t: is the register expire timeout in seconds

- proxy: SIP proxy
- realm: SIP realm
- authusr: auth (sipusername if separate extension id and authorization username have to be used)
- displayname
- transport (default/empty, UDP, TCP or TLS)

The Username parameter is mandatory, all the others are optional. JVoIP will use the defaults (if any) for the empty parameters.
Multiple accounts can be passed at once separated by semicolons (;).
An account must be specified as a SIP URI or as the following parameters separated by comma (,) :

Examples:
user:pwd@domain:port //a single extra account as a SIP URI
sip:john:secret@myserver.com:5060 // a single extra account as a SIP URI
"John Smith" <john:secret@myserver.com:5060> //a single extra account as a SIP URI specifying also the displayname
myserver.com,john,secret,180 //a single extra account as parameters
myserver.com:5061,john,secret,180, , , , ,TLS; //add a single extra account with TLS transport
john:jsecret@192.168.1.50:5060;kate:ksecret@192.168.1.50:5060 //multiple accounts as SIP URI's
92.168.1.50:5060,john,jsecret,180;92.168.1.50:5060,kate,ksecret;92.168.1.50:5060,mary,msecret,600 //multiple accounts as parameters with/without register interval

See the Multiple account registration FAQ for more details.

acceptsrvexpire

(int)
Accept the expires interval sent by the server.
0: no
1: yes (prioritize the contact expire)
2: yes (prioritize the global expire)
Default value is 1.

changesptoring

(int)
If to treat session progress (183) responses as ringing (180). This is useful because some IP-PBX never sends the ringing message, only a session progress and might start to send in-band ringing (or some announcement)
The following values are defined:
0: do nothing,
1: change status to ring
2: start local ring and be ready to accept media (which is usually a ringtone or announcement)
3: start media receive and playback (and media recording if the “earlymedia” parameter is set)
4: change status to ringing and start media receive and playback (and media recording if the “earlymedia” parameter is set to true)
Default value is 2.

*Note: on ringing status JVoIP is able to generate local ringtone. However this locally generated ringtone playback is stopped immediately when media is started to be received from the server (allowing the user to hear the server ringback tone or announcements)

allowcallredirect

(int)
Specify how to handle 301/302 answers for a INVITE requests.
Set to 1 to auto-redial on 301/302 call forward (will automatically launch a new call to the new target URI)
Set to 0 to disable auto call forward.
Default value is 1.
More details here.

natopenpackets

(int)
Change this option only if you have RTP setup issues with your server(s).
UDP packets to send to open the NAT device and initiate the RTP. Some servers will require at least 5 packets before starting to send the media after the 183 “session in progress” response. In this case set this value to 10 (In this way the server will receive at least 5 packets even on high packet loss networks)
0: no

1: write only an empty udp packet
2: write a normal RTP packet
3 or more: write this number of RTP packets

Default is 2

*Note: instead of sending more “fake” packets, you can set the “earlymedia” to 1 or more to begin the rtp stream immediately.
*Note: you can use the natopenpackettype parameter to specify the keep-alive RTP packet format:
-1 means auto, 1 means short CRLFCRLF packet (default). 2 means full RTP packet with zeroed content, 3 means ED-137 keep-alive.

earlymedia

(int)
Start to send media when session progress is received.
0: no
1: reserved
2: auto (will early open audio if wideband is enabled to check if supported)
3: just early open the audio
4: null packets only when sdp received
5: yes when sdp received
6: always forced yes
Default is 2.

*Note: For the early media to work, JVoIP has to open the NAT when SDP is received. This can be done by sending a few fake rtp packets or by starting to send the media immediately when session in progress is received. The first method consume less bandwidth, but it is not supported by some softswitch.

answerwithallcodec

(int)
Set to 1 to answer with a list of supported codec’s in the SDP by the UAS.
0: no (negotiate one final codec)
1: yes (list all supported/negotiated codec’s in 200 answer)
2: all (list all supported codec’s in 200 answer even those not offered by the caller)
Default is 0.

setfinalcodec

(int)
JVoIP might negotiate the final codec for a session with the ACK message if your server send multiple codec’s in the 200 OK call connect answer.
Some server cannot handle the final codec offer in the ACK message correctly.
In this case you will have to set this setting to 0, otherwise you will have one way audio.
0=never (RFC compliant)
1=auto guess (not send in case of certain servers and autocorrect in subsequent calls)
2=when multiple codecs are received
3=always reply with the final codec in the ACK message
Default value is 1.

ims3gpp

(int)
Enable 3GPP (IMS/5G/VoLTE) features.
Possible values:
-1: auto detect (default)
0: disable
1: basic
2: full/all

It is also possible to enable/disable certain 3GPP features separately. In this case use the following parameters instead of ims3gpp:

- ims3gpp_phonenum: use tel URI and set user=phone (-1: auto, 0: no, 1: user=phone, 2: also tel uri). Note: you might also use the sipproto param
- ims3gpp_minphonelen: tel uri only if phone number is longer then this. Default is 4. Set to 0 to use tel also for non-phone number peers
- ims3gpp_authuname: auth username must contain the domain (-1: auto, 0: no. 1: yes)
- ims3gpp_ussd: IMS USSD described at 3GPP TS 24.390. Send with API_SendUSSD, receive as USSD notifications (-1: auto, 0: no, 1: yes)
- ims3gpp_sms: +g.3gpp.smsip with MESSAGE using binary encoded PDU’s described in 3GPP TS 24.011 (-1: auto, 0: no, 1: yes)

- ims3gpp_smsc: SMSC Center Address number. Otherwise JVoIP might use the destination number (if that is a phone number)

enable_3pcc

(int)
Specify if to enable 3PCC Third Party Call Control as described at RFC 3725.
Note: 3PCC has nothing to do with 3GPP. They are completely different technologies.
This feature is often used in callcenters or for click-to-call to setup calls between two or more other parties.
Possible values:
0: disable (block 3PCC requests)
1: don't handle
2: accept (handle incoming requests)
3: initiate 3PCC calls by sending with no SDP
4: initiate 3PCC calls by sending SDP with connect IP set 0.0.0.0
5: initiate 3PCC calls by sending SDP with no media line
Default is 2.

If 3pcc is enabled, then JVoIP will send the codec answer in ACK if not received by INVITE or it will re-INVITE if received SDP with no media line or with 0.0.0.0 contact IP, thus allowing all the possible third party call controls.
Some more related information can be found here, or here.
You might also set the enableautoaccept parameter to 3 if JVoIP acts as an auto-responder.

ed137

(int)
Enable/disable the ED-137 behavior to be used for air traffic management services.
Set to 0 to ignore the ED-137 specification (act as normally)
Set to 1 to turn on the ED-137. This will change a list of internal settings and behaviors to conform with the ED-137 specification.
Default value is 0.
See the ED-137 guide for more details.

backupserver

(string)
Specify secondary SIP server address for failover in case if you have multiple SIP servers.
You can specify domain, ip and if port is not 5060 then you must also specify the port like mybackupserver.com:7080.
You might also specify a backupserverdomain parameter (use IP in backupserver and domain name in backupserverdomain).
More details can be found here.

inet

(int)
Internet protocol (IP layer).
Specify if you wish to prefer IPv4 or IPv6.
Possible values:
-1: Auto
0: IPv4 only (disable IPv6)
1: IPv4 (parse also IPv6)
2: Both (prefer IPv4)
3: Both (no preference)
4: Both (prefer IPv6)
5: IPv6 only (disable IPv4)

Default is -1/Auto (auto guessed from configurations such as serveraddress and from the SIP signaling / SDP)

Other IP related parameters (all are set based on this inet parameter, but can be set also separately if you have some special requirement):
hasipv4 (true/false), hasipv6 (true/false), listenonipv6 (true/false), ipvdualstack (true/false), cropipv6scopeid (true/false), preferipv (-1: auto (def to 1), 0: no preference, 1: prefer ipv4, 2: prefer ipv6)

proxyaddress

(string)
Outbound proxy address (Examples: mydomain.com, mydomain.com:5065, 10.20.30.40:5065)
Leave it empty if you don’t have a stateless proxy. (Use only the serveraddress parameter)
Default value is empty.

usehttpproxy

(int)
Used only for HTTP tunneling with Mizu VoIP servers.
0: no
1: same as sip proxy (proxyaddress)
2: system default
3: manual (must be set by the httpproxyurl parameter –deprecated after version 3.5)
4: auto
Default value is 4.

httpserveraddress

(string)
Useful when the transport parameter is set to 4 (auto) to specify the http tunneling gateway address.
Default value is null (address loaded from the “serveraddress” parameter)

transport

(int)
Transport protocol for the SIP signaling.
-1: Auto detect
0: UDP (User Datagram Protocol. The most commonly used transport for SIP)
1: TCP (signaling via TCP. RTP will remain on UDP)
2: TLS (encrypted signaling -SIPS)
3: HTTP tunneling (both signaling and media. Supported only by mizu server or mizu tunnel)
4: HTTP proxy connect (requires tunnel gateway or server)
5: Auto (automatic failover from UDP to HTTP as needed if tunnel gateway or server is used)
Default is -1.

See the How to configure SIPS FAQ point for more details about TLS transport.

mediaencryption

(int)
Media encryption method for the RTP streams.
-1: auto guess (might use SRTP if TLS transport is set)
0: not encrypted
1: auto (will encrypt if initiated by other party)
2: SRTP (recommended for RTP encryption)
Default is -1.

Note:
To enable full encryption, set both the transport and the mediaencryption parameter to 2.
Mode details here.

strictsrtp

(int)
Media encryption method.
0: most compatible (skip SRTP auth failures)
1: default auto (will try to use SRTP with less constrains, might failback to RTP on error or if SRTP is not supported by the peer)
2: strict (might disconnect if peer is not respect the standard or on protocol error)
3: allow only strict SRTP call, otherwise disconnect
Default: 1

Note:
This parameter is considered only if the mediaencryption parameter is also set.
Mode details here.

srtp_suite

(string)
You might specify the preferred srtp suite.

Possible values:
AES_CM_128_HMAC_SHA1_80 (recommended)
AES_CM_128_HMAC_SHA1_32 (supported)
AES_CM_256_HMAC_SHA1_80 (beta / not tested)
AES_CM_256_HMAC_SHA1_32 (beta / not tested)

Default: AES_CM_128_HMAC_SHA1_80

JVoIP might be able to negotiate also other values as suggested by your server.

video_config

(int)
Enable/disable video.
Set to 2 to disable video, to 4 to enable video on request or to 8 to enable video always.
All possible values:

- -1: auto (automatically activated on API requests)
- 2: never enable
- 4: on request only
- 6: ask (send offer, ask to enable)
- 8: yes (send/receive automatically)
- 10: always (force video send/accept)

Default is -1.

More details about video calls can be found here.

textmessaging

(int)
Specify text messaging mode (IM/chat/SMS/API)
-1: auto guess or ask (default)
0: disable all
1: disable incoming messages and auto guess outgoing mode
2: disable message sending and auto guess incoming mode
3: reserved
4: reserved
5: VoIP SMS
6: VoIP IM/chat (SIP MESSAGE)

Set the ims3gpp_sms parameter to 1 or 2 to use 3GPP binary encoded messages for SIP MESSAGE or to 0 to always disable

Note:

- *The old haschat and chatsms parameters are deprecated now but still supported*
- *JVoIP also has a built-in chat frame. You might enable/disable this with the displaychat parameter.*

dtmfmode

(int)
DTMF send method.
0=disabled
1=SIP INFO method (out-of-band in SIP signaling INFO messages)
2=auto (auto guess from peer advertised capabilities)
3=INFO + NTE
4=NTE (Named Telephone Events as specified in RFC 2833 and RFC 4733; DTMF in RTP headers)
5=In-Band (DTMF audio tones in the RTP stream)
6=INFO + InBand

Default is 2 (and you should change it only if you have a good reason to do so, since the auto-detect should work in all circumstances, except if your SIP server is sending bogus DTMF capabilities).

Note:
DTMF digits can be sent with the API_Dtmf or if you press a digit on the built-in user interface (if that is used).
When more than one method is used (dtmfmode 3 or 6), the receiver might receive duplicated dtmf digits.
If the message sending was initiated with SIP INFO, then it might failover to rfc2833 or inband if no answer or error code was received from server.

sendearlydtmf

(int)
Specify whether to allow sending DTMF digits before call connect
0=no
1=auto (yes if rfc2833 or inband is allowed and already sent/received rtp packets)
2=yes (always send)

Default: 1

offlinechat

(int)
Will try to resend not delivered messages later (on next register and/or when any message received from peer). The offline message queuing will take care of filtering out duplicates and will try to resend the message multiple times on failure until max number of resend or timeout reached.
-1=auto (usually yes, no for 3gpp sms)
0=no (disable offline chat)
1=yes (default)
2=force always

Default: -1

voicemail

(int)
Subscribe to voicemail notifications (MWI). Accepted values:
0=disabled
1=display voicemail only if NOTIFY is received automatically without subscription
2=auto-detect if voicemail SUBSCRIBE is needed
3=subscribe for voicemail messages after successful registration
4=subscribe for voicemail messages on startup
Default value is 2. You might set it to 3 if your server has support for MWI to be sure that JVoIP will check the voicemail in all circumstances.

See the MWI notification about incoming message waiting indicators.

voicemailnum

(String)
Specify the voicemail address. Most IP-PBX will automatically send the voicemail access number so you don't need to set this parameter.

transfertype

(int)
-1=default transfer type (same as 6)
0=call transfer is disabled
1=transfer immediately and disconnect with the A user when the Transf button is pressed and the number entered (unattended transfer)
2=transfer the call only when the second party is disconnected (attended transfer)
3=transfer the call when JVoIP is disconnected from the second party (attended transfer)
4=transfer the call when any party is disconnected except when the original caller was initiated the disconnect (attended transfer)
5=transfer the call when JVoIP is disconnected from the second party. Put the caller on hold during the call transfer (standard attended transfer)
6=transfer the call immediately with hold and watch for notifications (unattended transfer)
7=transfer with no hold and no disconnect (simplest unattended transfer)
8=transfer with conference (will put the parties to conference on transfer; will mute or hold the old party by default)

Default is -1 (which is the same as 6)

Note:

- *Unattended means simple immediate transfer (just a REFER message sent); Attended transfer means that there will be a consultation call first*
- *The most popular transfertypes are 1, 5, 6 and 7*
- *If you have any incompatibility issue, then set to 7 (unattended is the simplest way to transfer a call and all sip server and device should support it correctly)*
- *More details can be found here.*

transfwithreplace

(int)
Specify if replace should be used with transfer requests, so the old call (dialog) is not disconnected but just replaced.
This way the transferee and the transfer-target parties are not disconnected, just the other party is changed at runtime.
The feature is implemented by adding a Replaces= for the Contact header in the REFER request as described in RFC 5589 and RFC 5359.
The transferee (the party which receives the REFER transfer request) or the SIP server must be able to handle replaces for this to work.
Possible values:
-1=auto (if server/peer has replaces support and we are connected with the transfer target)
0=no
1=yes
2=force (use replaces even if peer doesn't have announce support or there is no session with the target)
Default is -1

Note: some SIP servers doesn't announce replace capabilities ("replace" tag into the Supported or Require SIP headers). In this can you will have to set the transfwithreplace parameter to 2.

allowreplace

(int)
Allow incoming replace requests when JVoIP is the transferee party (Handling the Replaces SIP header in incoming INVITE as described in RFC 3891).
0=no
1=yes and always disconnect old ep
2=yes and don't disconnect if in transfer
3=yes but never disconnect old ep
Default is 2.

replacetype

(int)
Specify how to handle incoming call transfer with replaces requests (How to handle the Replaces= Contact tag in incoming REFER requests)
-1=auto (defaults to 1)
0=disable
1=standard (sending the Replaces header in the new call INVITE request as described in RFC 3891)
2=in place (might be useful with servers without replaces support)
3=both (not recommended)

discontransfer

(int)
Specify if line should disconnect after transfer if not determined by the other transfer related configurations.
-1=auto
0=never
1=on C party connected status
2=on timeout
3=on connected or timeout
4=on ok for refer
Default is -1 (disconnect if transfertype is 1)

disconincomingrefer

(int)
Specify if line should disconnect after incoming transfer.
-1: auto
0: no
1: yes

Default is -1

inversetransfer

(int)
Specify inverse attended transfer.
0: no. The REFER will be sent to the ep for which the API_Transfer was called. This is the standard behavior.
1: yes. The REFER will be sent to the ep for which the API_Hangup was called. Might be used only if the original ep (for which the API_Transfer was called) doesn't support transfer (REFER sip message).
Default is 0 (standard transfer behavior)

transferdelay

(int)
Milliseconds to wait before sending REFER/INVITE while in transfer.
Default value is 400.

newdialogforrefer

(int)
Specify if the REFER have to be sent in a new dialog (for compatibility reasons).
0: no (after SIP standards)
1: yes, with no to tag
Default is 0

useserverdomainforrefer

(int)
Specify the domain part to be used in REFER for the Refer-To and Referred-By headers.
0: no (use default domains loaded from the dialog)
1: yes for Refer-To only
2: yes for Referred-By only
3: auto (usually yes for both)
4: force server domain for both
Default is 0.

Note: Servers should accept the domain from the dialog for the Refer-To and Referred-By headers after the SIP standards, however for some reasons a few popular server would reject the REFER request if they found some other domain (such as proxy server address) and thus this settings defaults to 3 for compatibility reasons.

checksubscriptionstate

(int)
Specify if to handle transfer subscription state (such as reload the call on transfer failure)
-1: auto
0: no
1: disconnect
2: reload

Default is -1

subscribefortransfer

(int)
Create subscribe dialog for call transfer.
-1: auto
0: never
1: if no notify received
2: always

Default is -1

enabledirectcalls

(int)
Specify whether to enable direct call to SIP URI (peer to peer or via server)
0: no (so you should use only the username part of the SIP URI to make calls. set the domain part as the sipserveraddress parameter)
1: check IP in URI (will recognize full SIP URI if the domain part can be resolved to a valid IP)
2: always check (so you can make calls to full SIP URI without a SIP server to be set)
3: crossdomain (so you can register to domain A and make direct call to domain B)

Default is 1

checksrvrecords

(int)
SRV DNS record lookup setting:
-1: auto (will check SRV record for the VoIP server, but remembers if fails and will not check again next time)
0: don't lookup (will use only A record)
1: lookup A records first. If fails then lookup srv record (because mostly the srv record is not set anyway)
2: lookup SRV records first for VoIP server address. If fails then lookup A record (RFC compliant)
3: always lookup SRV records first. If fails then lookup A records.
4: check also without the _sip._udp. prefix
If the SRV or A lookup returns multiple records, than SIP UA will failover to the next server on connection failures taking in consideration both record priority, weight and previous failures/successes.
Default value is -1.

dnslookup

(int)
Domain record lookup mode
0: auto (same as 2)
1: yes always re-query
2: use cache if needed (default)
3: use cache whenever possible
4: from cache only
5: disable
Default value is 0.

sendearlyack

(int)
Specify when to send ACK for 200 OK call connect
0: send the ACK only after successful RTP/SRTP and audio device initialization
1: send the ACK immediately on 200 OK receive

Default value is 0.

With the default 0 option the ACK might be send only with some little (10 msec – 2000 msec) delay which might cause the server to repeat the 200 OK answer. This is normal, but if for some reason you would like to eliminate this wait time, then set this parameter to 0 to send the ACK message immediately once the 200 OK arrives. When set to 0 then the call will switch to connected state even if audio/RTP fails (will disconnect on failure so on the server side you might see very short call duration for such failed calls with adequate disconnect reason)

useaudiodevicerecord

(boolean)
Set to false if you wish to disable audio recording from the local audio device.
You might set it to false if you wish to only receive the audio from the peer (not send) or if you will stream from an external source or from your app.
Default is true.

useaudiodeviceplayback

(boolean)
Set to false if you wish to disable audio playback on the local audio device.
You might set it to false if you wish to only send the audio from the peer (not receive) or if you will process the incoming audio stream yourself in your app.
Default is true.

enableaudiostreams

(int)
You can disable audio playback and/or recording with this option
0: disable all
1: disable recording
2: disable playback
3: enable all
Default value is 3.

audiodevicein

(string)
Audio device name for recording (microphone). Set to a valid device name or “Default” which would select the system default audio device.
The audio device can be set changed at runtime using the API_SetAudioDevice function.

audiodeviceout

(string)
Audio device name for playback (speaker). Set to a valid device name or “Default” which would select the system default audio device.
The audio device can be set changed at runtime using the API_SetAudioDevice function.

audiodevicering

(string)

Audio device name for ringtone. Set to a valid device name or “Default” which would select the system default audio device. You can also set it to “All” to have the ringtone played on all devices.
The audio device can be changed also at runtime using the API_SetAudioDevice function.

playring

(int)
Generate ringtone for incoming and outgoing calls.
0: no (you can generate ringtone also by using the Java API to playback a sound file when you receive ringing notifications)
1: play ringtone for incoming calls
2: play ringtone for incoming and outgoing calls. (ringtone for outgoing calls can be generated also by your VoIP sever. When remote ringtone is received, java softphone will stop the local ringtone playback immediately and starts to play the received ringtone or announcement)
Default is 2.

Note: you might set the earlymedia parameter to 5 in case if you wish to hear ringtone or any announcement from your server before call connect.

ringtone

(string)
Specify a ringtone sound file to be used. (Only the file name, not the full path. Copy the file near JVoIP.jar). If not specified, then JVoIP will use its own built-in ringtone for call alert.
The file should be in 8kHz 16 bit mono format: PCM SIGNED 8000.0 Hz (8 kHz) 16 bit mono (2 bytes/frame) in little-endian (128 kbits).
You can use any sound editor to convert your file to this format (usually from File menu -> Save as).

Possible values:

- Empty/default: will try to load the embedded ringtone or from the ring.wav near the app or in path
- File name (for example “myring.wav”): will try to load the specified file from near the app or from the java path (and failover to embedded ring if not exists)
- Full path (for example “C:\webphone\myring.wav”): will try to load the ringtone from the specified path first (and failover to embedded ring if not exists)

Default value is empty (which means that the default built-in “ring.wav” ringtone will be used).
Don’t use a “ring.wav” file if you wish to change the ringtone because in that case the webphone will load its built-in default. Set your file name to something else.

If you need a different ringtone for outgoing call, it can be set with the ringtoneout parameter. If the ringtoneout parameter is also set, then the ringtone will be used for incoming calls and the ringtoneout will be used for outgoing calls.

ringincall

(int)
Ring on incoming or outgoing call if we are already in a call.
-1: Auto
0: No
1: Only a beep for incoming call
2: Yes, normal ring
Default value is -1 (which means 0 if useaudiodeviceplayback is set to false, otherwise 1)

beeponincoming

(int)
Will play a short sound on incoming calls.
-1: Auto
0: No
1: If already in call
2. Always
Default value is -1 (which means 0 if useaudiodeviceplayback is set to false, otherwise 1)

Note: This is not the ringtone. This is similar to the ringincall setting but handled at SIP stack endpoint level.

beeptype

(int)
Specify beep method to be used if required.
-1: auto (1)
0: disable

1: system+dynamic
2: system+static
3: system
4: dynamic
5: static

Default value is -1 (which means 0 if useaudiodeviceplayback is set to false, otherwise 1)
The system beep can be also configured with the beepfrequency and the systembeeptype parameters (-1: auto, 0: disable, 1: system, 2: generate (duration), 3: both, 4+: generate duration milliseconds)

playdisc

(int)
Play a disconnect tone on call reject.
0: No
1: Auto (if there was no early media with possible disconnect reason announcement)
2: Always
3: Also for normal disconnect on connected call
Default value is 1

checkvolumesettings

(int)
Check if audio device is muted or volume settings are too low (and un-mute and/or increase volume if necessary).
0: no
1: at first run
2: always
Default value is 1

focusaudio

(int)
Specify whether to use audio "focus". (Auto decrease the volume of other apps).
This will use VoIP optimizations on windows (WAVE_MAPPED_DEFAULT_COMMUNICATION_DEVICE) and also will enable audio ducking (auto lowering the volume for other processes while JVoIP is in call).

Possible values:
0=auto (will default to yes on windows)
1=no
2=yes
Default value: 0

Note: This new setting will deprecate the old usecommdevice parameter, but usecommdevice can be still used as-is

volumein

(int)
Default microphone volume in percent from 0 to 100. 0 means muted. 100 means maximum volume, 0 is muted.
Default is 50% (not changed)

Note:
The result volume level might be affected by the AGC if it is enabled.
Volume levels above 70% might result in distorted sound.
You can also set/get the volume at runtime using the API_SetVolume/API_GetVolume functions.
You can also change the volume from OS system level volume controls.

volumeout

(int)
Default speaker volume in percent from 0 to 100. 0 means muted. 100 means maximum volume, 0 is muted.
Default is 50% (not changed)

Note:
The result volume level might be affected by the AGC if it is enabled.
Volume levels above 70% might result in distorted sound.
You can also set/get the volume at runtime using the API_SetVolume/API_GetVolume functions.
You can also change the volume from OS system level volume controls.

volumering

(int)
Default ringback volume in percent from 0 to 100. 0 means muted. 100 means maximum volume, 0 is muted.
Default is 50% (not changed)

Note:
Volume levels above 70% might result in distorted sound.
You can also set/get the volume at runtime using the API_SetVolume/API_GetVolume functions.
You can also change the volume from OS system level volume controls.

beeponconnect

(int)
Will play a short sound when calls are connected
0=Disabled
1=For auto accepted incoming calls
2=For incoming calls
3=For outgoing calls
4=For all calls
Default value is 0

agc

(int)
Automatic gain control.
0=Disabled
1=For recording only
2=Both for playback and recording
3=Guess
Default value is 3

This will also change the effect of the volumein and volumeout settings.
For the AGC to work the mediaench module must be also deployed. See the related FAQ section for more details.
Download: https://www.mizu-voip.com/Portals/0/Files/mediaench.zip

stereomode

(boolean)
Set to true for 2 audio channel or false for 1 (mono).
When stereo is set, the VoIP SDK will convert also mono sources to stereo output.
Default is false.

Note: many sound devices might convert mono sources to stereo by default

plc

(boolean)
Enable/disable packet loss concealment
Default is true (enabled)

vad

(int)
Enable/disable voice activity detection.
0: auto
1: no

2: yes for player (will help the jitter)
3: yes for recorder
4: yes for both
Default is 2.

Notes:

- The vad parameter is automatically set to 4 by default if the aec2 algorithm is used.
- If you wish to use VAD related statistics in your application, you might have to also set the vadstat parameter after your needs:
  0=no,1=auto (default),2=detect no mic audio,3=send statistics, 4=send also when changed. See the VAD notification and API_VAD for more details.
- If you wish to force vad (usually not required), then you might set the forcevad parameter accordingly:
  0: no (default), 1: force also if conference, 2: force also if rtp muted/holded, 3: force always
- If you want to disable audio related notifications (microphone warning on no signal detected) then set the enablenomicvoicewarning parameter to 0
- More details here

sipmsgnotifications

(int)
Enable/disable sending all received/sent SIP signaling messages as SIP notifications.
These might be useful only if you wish to extract some details which is not possible using the API functions.
Possible values:
0: no/disabled
1: yes/send notifications

Default is 0 (disabled)

rtpstat

(int)
Enable/disable RTP statistics by triggering RTPSTAT notifications.
Possible values:
-1: auto (will trigger RTPSTAT events in every 6-7 seconds and more frequently at the beginning of the calls)
0: disabled
Positive value: seconds to generate RTPSTAT events.

Default is 0 (disabled)
More details here.

aec

(int)
Enable/disable acoustic echo cancellation
0=no
1=yes except if headset is guessed
2=yes if supported
3=forced yes even if not supported (might result in unexpected errors)
Default is 1.

For this AEC to work the mediaench module must be also deployed. See the related FAQ section for more details.
Download: https://www.mizu-voip.com/Portals/0/Files/mediaench.zip
Note: the aec decision might be overwritten by the aectype parameter.

aec2

(int)
Secondary AEC algorithm.
0=no
1=auto
2=yes
3: yes with extra (this might be too much under normal circumstances)
Default is 1

Note: the aec decision might be overwritten by the aectype parameter.

aectype

(string)
AEC algorithm(s) to use.
One or more of the following strings separated by comma.
auto: will select automatically based on circumstances (CPU power, device capabilities, network)
none: disable aec
software: software aec (requires extra CPU processing)
hardware: hardware aec capabilities (not supported on all platforms)
fast: a fast software aec implementation
volume: this will just decrease the volume when speech detected from other end (using VAD)
Default is auto.

Note:
It is recommended to leave both the aec and aectype values with its default values.
To force full echo cancellation in all circumstances you might set the aec to "2", aec2 to 2 and the aectype to "software,hardware,fast"
To completely disable aec you might set the aec to "0", aec2 to "0" and the aectype to "none" (for example when you know that you will have only one way audio such as IVR calls)

denoise

(int)
Noise suppression.
0=Disabled
1=For recording only
2=Both for playback and recording
3=Auto guess
Default value is 3

For this to work the mediaench module must be also deployed. See the related FAQ section for more details.
Download: https://www.mizu-voip.com/Portals/0/Files/mediaench.zip

silencesupress

(int)
Enable/disable silence suppression
Usually not recommended unless your bandwidth is really bad and expensive.
-1=auto
0=no (disabled)
1=yes
Default is -1 (which means no, except mobile devices with low bandwidth)

rtcp

(boolean)
Enable/disable rtcp. (RFC 3550. Partial support)

codec

(string)
List of allowed audio codec's separated by comma.
Will accept one or more of the following strings separated by comma (upper or lower case doesn't matter):
pcmu, pcma, g711 (for both PCMU and PCMA), pcmu, pcma, g729, gsm, ilbc, speex, speexwb, speexuwb, opusnb, opuswb, opusuwb, opusswb, g7221, g7221uwb, def.

By default the SIP library will automatically choose the best codec depending on available codec's, circumstances (network/device) and peer capabilities.
Set this parameter only if you have some special requirements such as forcing a specific codec, regardless of the circumstances.
Default: empty (which means auto detection and negotiation)
Recommended value: leave it empty for "best" codec negotiation unless if you have some specific requirement.

Example: "OpusWB,G729,PCMU" //This will disable opus wideband, G.729 and PCMU
You can also set one single codec (if your server/peers allows only one codec or for testing to rule out all kind of codec negotiation related issues).
You can set to "def" to enable all the default codec's again.

Under normal default circumstances, the following is the built-in codec priority:

I. *Wideband Speex and Opus (These are set with top priority as they have the best quality. Likely used for VoIP to VoIP calls if the peer also has support for wideband)*
II. *G.729 (Usually the preferred codec for VoIP trunks used for mobile/landline calls because it's excellent compression/quality ratio for narrowband)*
III. *iLBC, GSM (If G.729 is not supported then these are good alternatives. iLBC has better characteristics and GSM is better supported by legacy hardware)*
IV. *G.711: PCMU and PCMA (Requires more bandwidth, but has the best narrowband quality)*

Note:

- Video codec's can be configured with the video_codec parameter. See the Video calls FAQ point for more details.
- G.722.1 is supported on Windows only
- For the codec configuration, either use the codec/prefcodec settings OR the use_xxx settings (where xxx is the codec name). You can achieve the same result with both. The use_xxx settings are a little bit more flexible regarding the codec prioritization.

prefcodec

(int)
Set your preferred audio codec.
Will accept one the following strings (upper or lower case doesn't matter):
pcmu, pcma, g711 (for both PCMU and PCMA), g729, gsm, ilbc, speex, speexwb, speexuwb, opusnb, opuswb, opusuwb, opusswb, g7221, g7221uwb
Default is empty which means the built-in optimal prioritization.
By default the SIP stack will present the codec list optimized regarding the circumstances (the combination of the followings):

- available client codec set (not all engines supports all codecs)
- server codec list (depending on your server, peer device or carrier)
- internal/external call: for IP to IP calls will prioritize wideband codecs if possible, while for outbound calls usually G.729 will be selected if available
- network quality (bandwidth, delay, packet-loss, jitter): for example iLBC is more tolerant to network problems if supported
- device CPU: some old mobile devices might not be able to handle high-complexity codec's such as opus or G.729. G711 and GSM has low computational costs

You can also fine-tune the codec settings with the use_xxx settings described below.

use_gsm

(int)
GSM codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1.

use_ilbc

(int)
iLBC codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1.

Note: the ilbc mode defaults to 30 and will be negotiated automatically. You might change the default with the "ilbcmode" parameter to 20 or 30.

use_speex

(int)
Narrowband speex codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

use_speexwb

(int)
Wideband speex codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 2

Note: to enable wideband in all circumstances, set the "disablewbforpstn" and "disablewbonmac" parameters to false.

use_speexuwb

(int)
Ultra wideband speex codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

Note: to enable wideband in all circumstances, set the "disablewbforpstn" and "disablewbonmac" parameters to false.

use_opusnb

(int)
Narrowband (8000 Hz) opus codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

Note:
The opx dll/so/jnilib files have to be placed near the JVoIP.jar for the opus codec to work.
The "use_opus" parameter is treated now as wideband instead of narrowband.

use_opuswb

(int)
Wideband (16000 Hz) opus codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 2

Note: to enable wideband in all circumstances, set the "disablewbforpstn" and "disablewbonmac" parameters to false. The opx dll/so/jnilib files have to be placed near the webhone.jar for the opus codec to work.

use_opusswb

(int)
Super wideband (24000 Hz) opus codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

Note: this codec might not work on all devices, depending on the audio device and driver capabilities!

use_opusuwb

(int)
Ultra wideband (fullband at 48000 Hz) opus codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

Note: to enable wideband in all circumstances, set the "disablewbforpstn" and "disablewbonmac" parameters to false. The opx dll/so/jnilib files have to be placed near the webhone.jar for the opus codec to work.

use_g7221

(int)
G.722.1 wideband codec.
Low-complexity Siren 7 audio coding format with sample-rate 16000 (7kHz audio bandwidth 16k samples/second).
Possible values: 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1.

Note:
The payload number is 114 by default and it will be negotiated automatically. You might change the default with the "g7221payload" parameter.
The bitrate is 24000 by default and it will be negotiated automatically. You might change the default with the "g7221bitrate" parameter to 24000 (24kbps) or 32000 (32kbps).
This codec is supported on Windows only.

use_g7221uwb

(int)
G.722.1 ultra-wideband codec.
Low-complexity Siren 7 audio coding format with sample-rate 32000 (14kHz audio bandwidth 32k samples/second).
Possible values: 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1.

Note:
The payload number is 115 by default and it will be negotiated automatically. You might change the default with the "g7221uwbpayload" parameter.
The bitrate is 32000 by default and it will be negotiated automatically. You might change the default with the "g7221uwbbitrate" parameter to 24000 (24kbps) or 32000 (32kbps) or 48000 (48kbps).
This codec is supported on Windows only.

disablewbonmac

(boolean)
Whether to disable wideband codec on mac devices.
Mac OS X has a JVM bug which prevents java to reopen the audio devices with different sample rate.
Set this to false only if you are using wideband codec for each calls (so there is no chance that a call have to be handled in narrowband)
Default value is true

disablewbforpstn

(int)
This setting will disable opus, speex and g.722.1 wideband and ultrawideband for outgoing calls to regular phone numbers since these are usually not supported for pstn calls and they might requires longer initialization.
0: no
1: check at first call
2: check all calls
Default is 1. Set to 0 to never disable wideband.

use_g729

(int)
G.729 codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 2
**In some countries a license/patent is required if you use G.729 so enable only if you have licenses or licenses are not required in your case (consult your lawyer if you are not sure)*

use_pcma

(int)
G711alaw codec. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 2

use_pcmu

(int)
G711ulaw codec. 0=never, 1=don't offer, 2=yes with low priority, 3=yes with high priority
Default is 1

alwaysallowlowcodec

(int)
Set to 2 to always put low computational and low bandwidth codec in the offer list, specifically GSM and PCMU. Low CPU or bandwidth devices might choose these codecs (such as a mobile phone on 3G).
Set to 0 to disable.
Default is 1 (auto)

codecframecount

(int)
Number of payloads in one UDP packet (frames per packet). This will directly influence RTP packet time (packetization interval) and packet size as shown here.

- By default it is set to 0 which means 2 frames for G729 and 1 frame for all other codec.
  This usually means 20 msec rtp packetization interval for all codec's and it is the most optimal setting, compatible with all SIP/media stack implementations.
- In case if you wish to minimize delay, then you might set the codecframecount to 1. This usually will result in 10 msec packetization interval and will increase the required bandwidth by 40% due to high header/data ratio.
- In case if you wish to minimize bandwidth, then you might set the codecframecount to 4.

mediatos

(int)
Sets traffic class or type-of-service octet in the IP header for packets sent from UDP socket which can be used to fine-tune the QoS in your network. As the underlying network implementation may ignore this value applications should consider it a hint.

The value must be between 0 and 255.
Valid values (HEX):

- 0: disabled

- 1: automatic (set to 10 under normal conditions and disabled when in tunneling)
- 2: low-cost routing
- 4: reliable routing
- 8: throughput optimized routing
- 10: low-delay routing
- or'ing the above values (from above 2)

Default value is 1.

Notes:
The old parameter name was udptos (still accepted, but changed to mediatos to clarify that this is applied only for RTP sockets, not for the signaling).

For Internet Protocol v4 the value consists of an number with precedence and TOS fields as detailed in RFC 1349. The TOS field is bitset created by bitwise-or'ing values such the following (DEC):

IPTOS_LOWCOST: 2
IPTOS_RELIABILITY: 4
IPTOS_THROUGHPUT: 8
IPTOS_LOWDELAY: 16

The last low order bit is always ignored as this corresponds to the MBZ (must be zero) bit.

This parameter might work only in preset environments; java JVoIP might not have enough rights to modify the IP TOS/DSCP headers.
Under Windows OS this has to be enabled by setting the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TcpIp\Parameters\DisableUserTOSSetting registry value to 0.

In case if you wish to set the TOS also for the signaling, use the sigtos parameter (the same way as the above mediatos, except that this will be applied for the signaling only, not for the media; it's default value is 1 which will default to 4 - reliable routing).

automute

(int)
Specify if other lines will be muted on new call
0=no (default)
1=on incoming call
2=on outgoing call
3=on incoming and outgoing calls
4=on other line button click
5=on outgoing call and on other line button click (like 2+4)
6=on any call and on other line button click (like 3+4)

Default is 0

autohold

(int)
Specify if other lines will be put on hold on new call
0=no (default)
1=on incoming call
2=on outgoing call
3=on incoming and outgoing calls
4=on other line button click
5=on outgoing call and on other line button click (like 2+4)
6=on any call and on other line button click (like 3+4)

Default is 0

holdontransfer

(int)
Specify if initial line should be put on hold on transfer.
-1=auto
0=no
1=yes, hold before transfer
2=yes, hold before transfer and reload if needed (on transfer failure)
3=yes, hold on successful transfer init (OK for REFER received or attended call progress or success received; transferred call might not be initiated/connected yet). Will reload if needed (on transfer failure).
Default is -1 (which means 3 for transfertype 5 and 6, otherwise 0)

unholdontransfer

(int)
Unhold the line just before the actual transfer if it was holded and the transfer is made with replaces (to prevent keeping the endpoint in hold)
-1=auto
0=no
1=yes with replaces if there was hold before
2=yes always
Default is -1 (which is the same like 1)

holdtype

(int)
Specify call hold type (which side to hold on API_Hold).
Call hold is usually initiated by the API_Hold function and with parameter you can specify which type of call hold do you wish to request.
Possible values:
-2: no
-1: auto (defaults to 2)
0: no (don't hold. a=sendrecv)
1: reserved (not used)
2: send only: instructs the peer to not send audio (mute speaker. Local JVoIP: a=sendonly, peer: a=recvonly)
3: receive only: not sending audio to the other (mute microphone. Local JVoIP: a=recvonly, peer: a=sendonly)
4: hold: both (mute both in/out audio. a=inactive)

Default is -1.

Note:

- *By default the hold will check the previous state. For example if previously it was local hold (2) and you switch to remote hold (3) then actually will switch to both hold.*
- *There is also a holdexplicite parameter which if set to 1, then the hold will be done strictly after the holdtype parameter, without considering previous state. This way you can also change between local and remote hold without the need to unhold first.*
- *This parameter was previously named "holdtypeonhold" which still works.*

muteonhold

(int)
Specify if call also have to be muted with hold (stop recording/playback and stop the according RTP stream).
Possible values:
-2: no mute,
0: mute in and out
1: mute out (speakers)
2: mute in (microphone)
3: mute in and out (same as 0)
4: mute default
5: according to call hold.

Default is 5.

Usually you should set this either to -2 or 5.

musiconhold

(int)
Specify if to play music on hold (MOH).
Possible values:
-1: auto (defaults to 0/disabled)
0: no/disabled
1: yes on hold with a=sendonly request initiated locally
2: yes on hold with a=recvonly received from the peer
Default is -1.

Note:

- *Option 1 (hold with* a=sendonly *initiated locally) is the recommended value if you wish to enable music on hold*
- *For MOH to work, you must copy a "music.wav" file near your app. You might use this file.*
  In case if you wish to use some other path for the audio file, then you can specify it with the musiconholdfile parameter.
  The file format should be PCM 8kHz 16 bit mono or as described at the API_PlaySound.
- *MOH might be played only if the holdtype is -1 (auto) or 2 (sendonly).*
- *MOH might be handled on the SIP server side instead.*

defmute

(int)
Default mute direction.
0: both
1: mute out (speakers)
2: mute in (microphone)
3: both
4: both (yes, again; not a typo)
5: disable mute

ackforauthrequest

(int)
If to send ACK for authentication requests (401,407).
0=no
1=yes (default)
Should be changed only if you have compatibility issues with the server used.

favorizecontactaddr

(int)
You may change it if you have compatibility issues with stateless proxies
0=never
1=no
2= conform RFC
3= yes. Sending for both server and contact URI (default)
4=always

prack

(boolean)
Enable 100rel (PRACK)
Set to false if you have incompatibility issues.
Default is false.

sendmac

(boolean)
Will send the client MAC address with all signaling message in the X-MAC header parameter.
Default value is false.

useragent

(string)
This will overwrite the default User-Agent setting.

Do not set this when used with mizu VoIP servers because the server detects extra capabilities by reading this header.

Default is empty.

alloweventsheader

(string)
This will overwrite the default Allow-Events setting.

Default value is: presence,refer,telephone-event,keep-alive,dialog

Note: not all capabilities are listed by default.

You might list the other expected capabilities here such as hold, talk, message-summary etc.

sendsessionid

(int)
Specify if to send the Session-ID header as specified in RFC 7989.

Possible values:

-1: if received (default)

0: no (disable)

1: yes (enable)

Default is -1.

customsipheader

(string)
Set a custom sip header (a line in the SIP signaling) that will be sent with all messages. Can be used for various integration purposes (for example for sending the http session id or any other data to your SIP server). Multiple lines can be separated by semicolon (;) or CRLF (\r\n).

Default value is empty.

Examples:
Add a SIP header:
customsipheader="X-MyData: 1"
Add multiple headers:
customsipheader="X-MyData: 1CRLFX-MyFlag: 2"
You can also rewrite known SIP headers if you set the keepoldcustomvalues parameter to -1:
customsipheader="To: <sip:user@domain.com>"

Notes:

- *You might use the keepoldcustomvalues parameter to control how the keys are merged: -1: merge also with SIP msg overwriting calculated known headers, 0: merge customsipheaders, but not with existing known precalculated headers (default), 1: don't merge / keep duplicated headers.*
- *Custom SIP headers can be set also at runtime with the API_SetSIPHeader function.*
- *Instead of custom SIP headers you might use other method to communicate with your backend, such as NOTIFY, INFO, UUI, MESSAGE, UUSD, custom SIP requests or separate API calls.*

customsdpfield

(string)
The customsdpfield works like the customsipheader parameter but it is to be used for SDP (or for any SIP message body) instead of the SIP headers.
Use the customsdpfield parameter to set a custom global SDP field (a line in the SDP body before the m= line) that will be sent with all messages.
It can be set also at runtime with the API_SetSDPField function.

customsdpmediafield

(string)
Set a custom media SDP field (a line in the SDP body after the m= line) that will be sent with all messages.
It can be set also at runtime with the API_SetSDPField function.
Default value is empty. Multiple lines can be separated by semicolon (;).

rtpextraheader

(string)
Set RTP extra header bytes.
The string will be converted to RTP extra header word(s).
If only one extra word needs to be set, then pass it as an integer (integer value converted to string).
Multiple words can be set by separating them with semicolon (;) .Example: 98;76543 will set the first word to 98 and the second word to 76543.
It can be set also at runtime with the API_RTPHeaderExtension function "extension" parameter.
See the RTP header extension FAQ point for more details.

rtpextraheader_profile

(int)
Set the profile number (the first two bytes) for the RTP extra header if required.
If not set then the profile bytes will be set to 0.
It can be set also at runtime with the API_RTPHeaderExtension function "profile" parameter.
See the RTP header extension FAQ point for more details.

sip_uui

(string)
Specify UUI data (User-to-User Call Control Information as described in RFC 7433).
Can be used to send data to other endpoints. Not all endpoints and SIP servers supports this feature.
*If prefixed with *0: then the UUI will be sent only to peer with the User-to-User header.*
*If prefixed with *1: then the UUI will be sent only to target party with call transfer or redirect escaped in Contact or Refer-To URI.*
*If not prefixed or prefixed with *2: the UUI will be sent both way.*

It can be set also at runtime with the API_SetUUI function.
Default value is empty.

Simple example: mydata
Example with parameters: *2:mydata;encoding=hex;purpose=foo;content=bar

techprefix

(string)
Add any prefix for the called numbers.
Default is empty.

normalizenumber

(int)
Normalize (called) numbers by removing .-,;()[]: and space if the string otherwise doesn't contains a-z or A-Z characters (looks like a phone number).
Possible values:
-2: never
-1: auto (usually defaults to 1 yes, except if our username also contains special characters)
0: no
1: yes
Default is -1.

numrewriterules

(string)
Simple called number prefix rewrite rules.
Although you can rewrite any number from your own code as you wish, you might use this numrewriterules parameter to let the SIP stack rewrite the prefix of the called numbers.
Format (parameters): from;to;minlength;maxlength
If you wish to add more rewrite rules, then you can separate them with comma.
Example:
07;00407;8;12,74;004074
This will rewrite prefix 07 to 00407 if number length is between 8 and 12 characters and will rewrite prefix 74 to 004074 regardless of the number length.

Note: this parameter is same with numpxrewrite

numrewriterulesadv

(string)
Advanced number rewrite rules.
Although you can rewrite any number from your own code as you wish, you might use this numrewriterulesadv parameter to let the SIP stack rewrite numbers.

Parameters can be separated by ; or _P_
Rules can be separated by , or _L_
Rules must begin with the separator character.

Parameters:
1. apply to sessions: 0=outgoing calls, 1=incoming calls, 2=all calls, 10=outgoing all, 11=incoming all, 12=all (Defaults to 0)
2. apply if number: 0=all numbers, 1=start with, 2=ends with, 3=contains, 4=equals, 5=not contains, 6=not start with (Defaults to 1)
3. string (used if needed for the above condition)
4. min number length (Defaults to 7)
5. action: 0=rewrite, 1=rewrite all, 2=remove, 3=add prefix, 4=add suffix / 10=auto answer, 11=ignore, 12=forward, 13=reject (Defaults to 2)
6. rewrite from (used if the above action is 0)
7. string (used if needed for the above action is less then 10)
8. stop further rule processing if this rule match (0/1) (Defaults to 0)

Example rule to insert 00 prefix if not already there for outgoing calls if number length is more than 8 digit:
_P_0_P_6_P_00_P_8_P_3_P_0_P_00_P_0_P__L_

mustconnect

(boolean)
If set to true, than users must register before to make any calls.
Default value is false.

rejectonbusy

(boolean)
Set to true to reject all incoming call if there is already a call in progress.
Default value is false.

disablesamecall

(int)
This setting can be used to don't allow/block double call the same number.
Possible values:
-1: auto-guess depending on config/usage
0: allow multiple calls to the same destination username/number.
1: reject call attempt with numbers already in call when the previous call was initiated less than 6 seconds ago.
2: reject call attempt with numbers already in call.
3: reject call attempt with numbers already in call regardless if it is incoming or outgoing current call.

Default value is 1

maxsimcalls

(int)
Maximum number of simultaneous calls (channel limit/number of call limit).
Will impose a max concurrent call limit applied for both incoming and outgoing calls.
For example if you set it to 1, then the enduser can't initiate or receive new calls while already in call.
Default value is -1 which means no call limit.

Note: if set to 0 then all calls will be denied.

redialonfail

(int)
Retry the call on failure or no response.

- 0: no
- 1: yes

Default value is 1.

A related setting is the "allowrecall" parameter which can be used for more precise control:

- *0: no*
- *1: normal invite resend (same as redialonfail 0)*
- *2: yes srv record and other important retry*
- *3: always when malfunction is detected such as no answer or no incoming RTP (same as redialonfail 1)*

These parameters will affect only failed unconnected calls.

autoredial

(int)
Redial on call disconnect.
Possible values:

- 0: no (default)
- 1: if it was disconnected because rtp timeout and we were the originator
- 2: if it was disconnected because rtp timeout
- 3: if we were the originator
- 4: if we were the called
- 5: always

The redial will be performed only if there are no other call in progress and the call was not locally terminated (such as pushing the Hangup button or calling the API_Reject or API_Hangup functions).
For better control on when to redial, you might use the API_Call instead on certain conditions as required for your app logic.

callforwardonbusy

(String)
Specify a number where calls should be forwarded when the user is already in a call. (Otherwise the new call alert will be displayed for the user or a message will be sent on the API)
Default is empty.
More details here.

callforwardonnoanswer

(String)
Forward incoming calls to this number if not accepted or rejected within 15 seconds. You can modify this default 15 second timeout with the callforwardonnoanswertimeout setting.
Default is empty.
More details here.

callforwardalways

(String)
Specify a number where ALL calls should be forwarded.
Default is empty.
More details here.

calltransferalways

(String)
Specify a number where ALL calls should be transferred.
This might be used if your softswitch doesn't support call forward (302 answers).
Default is empty.
More details here.

autoignore

(int)
Set to ignore all incoming calls.
0=don't ignore
1=silently ignore (this will not reject the calls, but they will be ignored/muted)
2=reject (auto reject)
Default value is 0.

Note: If the hideautocalls is set to 1 and the autoignore is 2, then there will be no status reports about the rejected calls.

autoaccept

(boolean)
Deprecated by enableautoaccept.
Enable/disable automatically accepting incoming calls.
Set to true to automatically connect all incoming calls (auto answer).
Default is false.

You might also specify a delay with the autoacceptdelay parameter.
More details here.

enableautoaccept

(int)
Specify to enable auto-answer for incoming calls.
Possible values:
0: never (except for barge-in calls if the bargeinheader is set and it has a match in the incoming call INVITE message)
1: enable only if the autoacceptheader is set and it has a match in the incoming call INVITE message
2: enable also for server-side initiated auto answer (if headers such as Call-Info, Alert-Info, Answer-Mode or Auto-Answer are received with the incoming INVITE)
3: force auto answer: auto connect all incoming calls
Default is 1.

For example you might just set this parameter to 3 to auto-answer all incoming calls.
More details here.

Note: the old autoaccept parameter is deprecated now and replaced with this parameter (enableautoaccept set to 3 is the same like the old autoaccept set to true).

autoacceptheader

(string)
With this parameter you can specify a special string which when received form your SIP server incoming call INVITE message, JVoIP will auto-answer the incoming call.
For example if you set it to "myspecialstring" and the "myspecialstring" is a substring of the incoming INVITE, then the call will be automatically accepted.
The string can be set to anything, including an extra header (such as "X-My-Auto-Accept: yes") or even to a caller-id.
The search is case insensitive.
Default is empty (which means that it will not check for any strings for auto-accept)

Note: Instead of a special string, you might set the enableautoaccept parameter to 2 and use the known server-initiated headers instead such as Call-Info, Alert-Info or Auto-Answer.
More details here.

bargeinheader

(string)
JVoIP barge-in feature allows you to bare into any call and create a hidden conference endpoint, so you can hear the JVoIP conversation(s).
With this parameter you can specify a special string which when received form your SIP server incoming call INVITE message, JVoIP will auto accept the incoming call as a hidden call-leg. This is very similar to the above autoacceptheader parameter. The main difference is that in this case the incoming call will be hidden.
This feature can be used to silently listen for JVoIP calls. It is often used in callcenters by supervisors to monitor the agents activity.
Default is empty (no barge-in calls checked and allowed).

Note:
In case if you wish to use JVoIP for the supervisor barge-in calls, then you can set it with the customsipheader parameter. For example if you have set the bargeinheader for the agents JVoIP instances to "X-BargeIn: yes", then set the exact same as the customsipheader parameter for the supervisor JVoIP instance and just call any agent(s) to listen on their calls.
Instead of using barge-in calls, you might consider to just enable call recording or use media streaming as alternatives for quality controls.

autoacceptdelay

(int)
Specify a delay (milliseconds) for call auto-answer.
This is applicable for both server side initiated auto-answer or if you have set the enableutoaccept parameter to 3 to auto answer all incoming calls.
Possible values:
-1: default (will check the answer-after flag received from your server or instant answer if not received)
0: always instant answer
1+: always force this amount of delay in milliseconds
Default is -1.

This timer has a resolution of around 1000 milliseconds (1 seconds).
For example if you set this to 4000 then then calls will be answered only after a 4 seconds delay (up to 5 seconds).
More details here.

blacklist

(string)
Block incoming communication from these users (users/numbers/IP addresses separated by comma).
Default value is empty.

whitelist

(string)
Allow incoming SIP requests only from these users (users/numbers separated by comma).
Be aware that your server might send requests on it's own, such as OPTION requests. In this case the username used by the server should be also allowed, otherwise such requests will be ignored.
Default value is empty.

blockmode

(int)
Specify how to apply the blacklist and the whitelist.
Possible values:
0: disable (black/white list will be applied only for presence and BLF)
1: ignore messages at transport level
2: reject call sessions
3: both
Default: 1

rejectcallto

(int)
Set to ignore calls if target doesn't match
0=accept all incoming calls
1=check if target user match (reject sessions for other users)
2=check rinstance
3=check rinstance strict
4=check all strict
Default value is 0.

hideautocall

(int)
Set to 1 to suppress notifications (STATUS, CDR) from automatically handled calls (ignored, forwarded, rejected and similar).
0=send status notifications also about auto handled calls
1=do not send status notifications from auto handled calls
Default is 0.

ringtimeout

(int)
Maximum ring time allowed in millisecond.
Set to 0 to disable.
Default is 90000 (90 second)

calltimeout

(int)
Maximum speech time allowed in millisecond.
(With other words, this is a maximum call duration limitation applied for each call session separately).
Set to 0 to disable.
Default is 10800000 (3 hours)

startsipstack

(int)
Automatically start the sipstack after a specified time.
0=no (the sipstack is to be started by the API_Start function or it will be started on the first register or call attempt)
1=on startup if serveraddress/username/password are set (the sipstack will be started at app init)
2=on startup always (the sipstack will be started at app init)
Other=seconds (the sipstack will be started after the specified seconds)
Default value is 1

You can set to 0 if there is less change that JVoIP will be used once the users will open the app or webpage hosting JVoIP.
You can set to 1 or higher if there is a high probability that the user will use JVoIP to make calls (this will shorten the setup time for the first call).

If you are using JVoIP as a library, then you can also disable SIP stack start by using the webphone(int startsipstack) constructor passing 0 as the startsipstack parameter. Example: webphone wobj = new webphone(0);
You can also start the SIP stack explicitly by using the API_Start function.

timer

(int)
You can slow down or speed up the SIP protocol timers with this setting. You may set it to 15 if you have a slow server or slow network.
Default value is 10.
Note: This is just a relative value, not milliseconds. The exact timing will be calculated based on many other circumstances, but you can influence it by changing this value. For example set to 20 to double the interval.

timer2

(int)
Same as "timer" but it affects idle, connect and ring timeout and maximum call durations.
Default value is 10.

Note: This is just a relative value, not milliseconds. The exact timing will be calculated based on many other circumstances, but you can influence it by changing this value. For example set to 20 to double the interval.

timer3

(int)
Message retransmission timer (SIP request resend on no answer).
Default value is 10.

Note: This is just a relative value, not milliseconds. The exact timing will be calculated based on many other circumstances, but you can influence it by changing this value. For example set to 20 to double the interval.

maintimer

(int)
Main thread timer to handle synchronized events.
Default is -1 which means auto calculated (usually 100 milliseconds)

conference_addalllines

(int)
Always add all lines to conference on API_Conf, regardless of the requested line.
Possible values:
-1: auto guess (default; will be auto set to 1 if confline parameter is set for API_Conf)
0: no (add only requested lines to conference)
1: yes (always add all lines to conference calls)

conference_tracklines

(int)
Required for multiple simultaneous conference calls to track other endpoints (othereplist) involved in a conference. Otherwise only one conference is possible at a given time.
Possible values:
-1: auto guess (default; will be auto set to 1 if confline parameter is set for API_Conf)
0: no (don't track; only one conference is possible)
1: yes (track; multiple simultaneous conference calls are possible)

mediatimeout

(int)
RTP timeout in seconds to protect again dead sessions.
Calls will be disconnected if no media packet is sent and received for this interval.
You might increase the value if you expect long call hold or one way audio periods.
Set to 0 to disable call cut off on no media.
Default value is 300 (5 minute).

At the beginning of the calls, the half of the mediatimeout value is applied (2.5 minute by default if there was no incoming audio at all).

mediatimeout_notify

(int)
RTP timeout in seconds for API notify.
After this timeout a warning message is sent via notifications without any further action.
The following log will be generated: “WARNING,media timeout (notify)”
Default value is 0 (disabled)

rtpkeepaliveival

(int)
RTP stream keep-alive packet send interval in milliseconds.
This is useful if your PBX has an RTP timeout setting to prevent disconnects when the java softphone is hold or muted.
Default value is 25000 which means keep-alive packets in every 25 seconds on hold/mute/silence.

sendrtponmuted

(boolean)
Send rtp even if muted (zeroed packets)
Set to true only if your SIP server is malfunctioning when no RTP is received (such as dropping the call on media timeout).
Default value is false.

sendrtponfailed

(int)
Specify if to send RTP packets if unable to open the recording device.
Possible values:
0: No
1: At the beginning (to open the NAT for incoming audio)
2: Yes always
3: Must
Default is 1.

sendrtpondisabled

(int)
Specify if to send RTP packets if useaudiodevicerecord is set to false.
Possible values:
0: No
1: at the beginning (to open the NAT for incoming audio)
2: yes always
3: must
Default is 3.

discmode

(int)
For call disconnect compatibility improvements. Some VoIP devices might have bugs with CANCEL forking, so it is better to always send a BYE after the CANCEL message on call disconnect. In this case set the discmode parameter to 3.
1: quick
2: conform the RFC
3: send BYE after CANCEL when needed
4: double: always repeat the CANCEL and the BYE messages
Default value is 2.

waitforunregister

(int)
Maximum time in milliseconds to wait for unregistration when the API_Unregister is called or the java sip stack is closed.
If set to 0 that an unregister message is sent (REGISTER with Expires set to 0) but JVoIP is not waiting for the response, which means that it will not repeat the un-register in case if the UDP packet was lost.

Default value is 2000.

clearcredentialsonunreg

(int)
Specify whether user login details (server/username/password settings) have to be cleared on unregister or not.
Possible values:
-1: auto
0: no
1: yes
Default: -1

md5

(string)
Instead of using the password parameter you can pass an MD5 checksum for better protection: MD5(username:realm:password)
(The parameters are separated with the ':' character)
The realm is usually your server domain name or IP address (otherwise it is set on your server)
If you are not sure, you can find out the realm in the "Authenticate" headers sent by your server with the" 401 Unauthorized" messages. Example:
WWW-Authenticate: Digest realm="YOURREALM", nonce="xxx", stale=FALSE, algorithm=MD5
Default is empty.

realm

(string)
Set if your server realm (SIP domain) is not the same with the "serveraddress" parameter.
If the "md5" parameter was set, then this must match with the realm used to calculate the md5 checksum.
Default is empty, which means that the "serveraddress" will be used.

encrypted

(boolean)
Specify if the transport will be encrypted (both media and the signaling)
Compatible only with Mizu VoIP servers.
Automatically turned on when using http tunneling.
Default is false.

authtype

(int)
Some IP-PBX doesn't allow "web" or "proxy" authentication.
0=normal/auto (default)
1=only proxy auth
2=only simple auth

sipproto

(string)
You can change the URI with this string.
Default value is "sip".
You might change it to "sips" or "tel" if needed.

sipusername

(string)
Specify default SIP username for authentication. Otherwise the "username" parameter will be used for both the username and the authorization name.
Default is empty.

If this is not specified, then the "username" will be used for the From field and also for the authentication.
If both the username and sipusername is set then:
-the username will be used in the From and Contact fields (extension ID/CLI/caller-id)
-the sipusername will be used for authentication only (authentication username)

More clarifications here.

displayname

(string)
Specify default display name used in “from” and “contact” headers.
This is usually the full name of the enduser such as “John Smith”.
Default is empty (the “username” field will be displayed for the peers)

More clarifications here.

pwdencrypted

(int)
Specify if you will supply encrypted passwords via parameters or via the Java API
0=no (default)
1=xor
2=des+base64
3=xor+base64 (this is the preferred method; easiest but still secure enough)
4= base64
This method is deprecated from version 3.4. All parameters can be passed encrypted now by just prefixing them with the “encrypted__X__” string where X means the id of the encryption method used.

From version 4.8 there is no need to specify this parameter anymore. Just prefix any parameter with encrypted_X as described here.

voicerecording

(int)
0=no (default)
1=local
2=remote ftp or http upload
3=both

If set to 2 or 3 then either the voicerecftp_addr or the http_addr parameter have to be set.
Local recorded files are placed in the mwphonedata folder which is usually created near the JVoIP.jar if it has permission to its own folder or otherwise to another location such as the user home directory.
The voice recording can be also toggled on/off at runtime with the API_VoiceRecord function.
For more details see the voice record FAQ.

voicerecfilename

(int)
The format of the recorded filenames.
0=date-time + peer name (default)
1=date-time + sip call-id
2=sip call-id
3=date-time + username
4=date-time + username + peer name

The date-time will be formatted in the following way: yyyyMMddhhmmss

Note: You can also use the “voicerecfilenameprefix” parameter to add a prefix for the file name.

voicerecftp_addr

(string)
FTP location for the recorded voice files if the “voicerecording” parameter is set to 2 or 3.
Format: ftp://USER:PASS@HOST:PORT/PATH/TO/THEFILE
Example: ftp://user01:pass1234@ftp.foo.com/FILENAME

The FILENAME part of the string will be replaced with the file name according to the “voicerecfilename” parameter.

voicerecformat

(int)
Recorded file compression.
0: PCM wave stereo files with separate channels for in/our (default)
1: raw gsm. (two files will be generated for each call. One for the recorder file and another for the playback. These files can be played with players supporting gsm codecs for example quicktime, which works also as a browser plugin, or a winamp plugin is downloadable from here. Backups here.)
2: ogg/vorbis (optional, on request; module not included by default; notify Mizutech to include ogg/vorbis support in your build if you need this option)
3: mp3 (Make sure that lame is near the jar or is found on the path. It can be downloaded from here for windows or use your package manager on linux. New JVoIP versions will try to download it automatically on demand)

voicerecordingbuff

(int)
The maximum recorded file length.
-1: dynamic, no limit
1: max around 1 minute
2: max around 2 minute
…

Default is -1.

syncvoicerec

(int)
How to synchronize the recording/playback side:
-1: Auto
0: No (don't synchronize)
1: Yes (fill with noise the other channel)
2: Yes (wait for both side)
Default: 2

Note: you should set this to 0 if you have one-way audio calls such as IVR calls.

uploadretry

(int)
Specify whether the file upload should be retried on failure if the voicerecording parameter is set to 2 or 3.
0: no
1: once
2: until success
Default: 1
(Old parameter now was ftpretry which can be still used but now applies also for http uploads)

ftp_addr

(string)
FTP location for general storage (for example for settings, contactlists)
Format: ftp://USER:PASS@HOST:PORT/PATH/TO/THEFILE
Example: ftp://user01:pass1234@ftp.foo.com/FILENAME

The FILENAME part of the string will be replaced with the actual file name.

http_addr

(string)
HTTP location for general storage (for example for voice recording, settings,or contactlists). It can be a HTTP or HTTPS URL.
Example: https://www.yourdomain.com/storage/

For this parameter to work, you need a server side script capable to accept the file uploads (standard HTTP file upload).

If the file name part can be omitted (in this case the filename will be set as specified by the voicerecfilename or API_VoiceRecord function call).

You can also suggest a particular file format by appending its extension to the file name (for example .wav or .mp3).
For example: http://www.foo.com/myfilehandler.php?filename=callrecord_DATETIME_USER.wav

You can also set just a path, without any URL query parameter, like this:
For example: https://www.foo.com/myfilehandler.php/rec_DATETIME_CALLID

Example HTTP POST header packet if you set the url to "http://yourdomain.com/myapi/voicerecord/anyentry?filename=callrecord_DATE_CALLID.mp3":

```
POST /myapi/voicerecord/anyentry?filename=callrecord_2024_02_12_xxx.mp3 HTTP/1.1
Host: yourdomain.com
User-Agent: webphone
Accept: */*
X-type: fileupload
X-filename: callrecord_2024_02_12_xxx.mp3
X-user: local_username
X-caller: caller_party
X-called: called_party
X-callid: sip_callid
X-server: your_sip_server_address
Content-Length: 18623
Expect: 100-continue
Content-Type: multipart/form-data; boundary=--------------------------fde9999399e1b8eb

--------------------------fde9999399e1b8eb
Content-Disposition: form-data; name="file"; filename="callrecord_2024_02_12_xxx.mp3"
Content-Type: application/octet-stream
.....
```

You will receive "file" as the form name parameter and the name of the file as the form data "filename" parameter.
(So you will receive the file name as both the "filename" form parameter and also in the X-filename HTTP header).
The content type can be application/octet-stream, audio/x-wav, audio/mpeg, audio/x-gsm or audio/ogg.
(Regardless of the suggested file name, you can save the files on your server with any name, this is your choice).
A working example for PHP can be downloaded from here.
More details about handling HTTP file upload: C#, ASP.NET, PHP, PHP (2), NodeJS, Java.

autocfgsave

(int)
Configurations and statistics are stored in a local file to be reused in next sessions.
This is not critical and the Java VoIP client will work just fine if this file is lost or deleted by the user.
Sometime is useful to not allow configuration/settings storage on the user device.
The autocfgsave option can be set to the following values:

- -2: disable all file write forced
- -1: disable file write
- 0: disable config storage
- 1: save only
- 2: load only
- 3: save and load

Default is 3.

configversion

(int)
Set to a positive value or increment it with one to reset setting once (clear all previously stored or cached settings).
Default is 0 which means no reset.

resetsettings

(boolean)
Set to true to clear all previously stored or cached settings.
Should be passed as command line parameter only.
Default is false.

Note: If set to true, then JVoIP will not remember any previous settings, including its own internal optimizations, DNS caches and NAT related discoveries.

maxlogsize

(int)
Specify maximum log file size in bytes.
Default is 31457280 (which means 30 MB).

Note:
This is just an upper hard limit and it is relevant only if you run JVoIP for days without restart with higher loglevel. Otherwise, it is unlikely that you will reach it since the logs are cleared with each restart (as specified with the above deloldlogs parameter). Under normal circumstances the log file is usually below 10 MB with loglevel 5 and below 100 KB with loglevel 1.

signalingport

(int)
Specify the local SIP signaling port to use (bind port / local port). This might be useful for some use cases, for example if you wish to reach JVoIP directly, without the need to register to a SIP server first. See the P2P FAQ point about this.
Default value is 0 (which means that a stable port will be selected randomly at the first usage and will be kept the same)

Note:
This is not the port of your server where the messages should be sent. This is the local port for sip user agent and it doesn't have to be 5060. For more details see here.
If JVoIP can't bind to the specified port (for example if the port is already used by some other local app) then it will try to use another port.
In some circumstances (when new connections are required or cannot bind) JVoIP might use additional local ports above the specified signalinport value (usually up to 10, until signalingport + 10).

rtpport

(int)
Specify local RTP port base.
Default is 0 (which means signalingport + 2)
Note: If not specified, then VoIP SDK will choose signalingport + 2 which is then remembered at the first successful call and reused next time (stable rtp port). If there are multiple simultaneous calls then it will choose the next even number, thus the port range will be from: rtpport to: rtpport + 2x maximum number of calls + any additional ports which might have been skipped if used by some other app.
The RTCP port for each call will be the rtpport+1.
For video (if enabled) JVoIP will use the rtpport+10 port number by default.
For more details see here.

incrtpport

(int)
Increment RTP port for each call by this value.
Might be needed only with some misbehaving routers. If not set, then JVoIP will try to use the same RTP ports for all calls if available (if can bind to it, otherwise will try the next even number).
Default is 0.

Note: You will still have a different RTP port for each simultaneous calls even if this is set to 0.

bindip

(String)
Specify local network interface IP address for the sockets to bind to.
This parameter might be used only on devices with multiple local IP addresses to force the specified IP.
This parameter should be used only with local private IP (an IP address which is present on the device).
Default is empty (by default it doesn't bind to any IP and it is up to the OS routing table from which IP the packets are sent)

Note: This parameter should be used only in very specific circumstances when the device has multiple IP address and you wish to use only one of them.

localip

(String)
Specify local IP address to be used for the SIP signaling.
This parameter might be used only on devices with multiple ethernet interface to force the specified IP or if JVoIP is behind NAT to specify its public IP.
This parameter should be used only with static IP (if the device IP doesn't change dynamically from DHCP) or if you can better detect the best IP to be used in your app instead of letting JVoIP to auto detect it.
Default is empty (auto-detect best interface to be used or detect the external IP)

Note:
JVoIP by default will auto detect the "best" IP to be used and this parameter should be used only in very specific circumstances.
It is also possible to set the address sent in SDP explicitly with the localsdpip and localsdpport parameters.

favlocalip

(String)
Specify local subnet preference (favor local IP).
For example if the device where JVoIP is running might have two separate IP (such as 192.168.1.5 and 10.0.0.5) then you might set the favlocalip to 192 to always prefer the 192.x.x.x subnet.
This parameter might be used only on devices running on a known environment (local LAN) in case if you wish to suggest the subnet to be used.
Default is empty (auto-detect best subnet to be used)

Note: JVoIP by default will auto detect the "best" IP to be used and this parameter should be used only in very specific circumstances.

bindtolocalip

(int)
Specify if sockets must bind to the configured (localip parameter) or to the auto detected local ip.
Possible values:
-1: auto (guess)
0: no
1: yes
Default is 0.

Note:

- *This setting can applied even if the localip parameter is not explicitly set (auto detected best local ip to use), but it doesn't make sense if bindip is set.*
- *The disadvantage of setting to 1/yes is that the SIP stack might not tolerate so easy the local IP change (in case if running on a dynamic IP assigned DHCP)*
- *The advantage of setting to 1/yes is that the packets will be always sent from the same address specified in the signaling (some SIP server might be confused if receives packets from a different address then the address specified in the signaling)*
- *The -1/auto will bind to local IP if detected or configured local IP is found locally assigned and it is in the same subnet with the SIP server*
- *JVoIP by default will auto detect the "best" IP to be used and this parameter should be used only in very specific circumstances.*

sockbuffersize

(int)
Specify the receive/send buffer size in bytes for the sockets (SO_RCVBUF/ SO_SNDBUF).
The default is –1 which means that the buffer sizes will be auto calculated based on the socket type (signaling, audio RTP, video RTP, RTCP) and other factors.
Usually the following values are used by default (but the exact value might be different depending on factors such as TCP vs UDP, maxlineex, video usage, etc):

- Signaling: 64000
- RTP Audio: 40000
- RTP Video: 600000
- RTCP: 8000
- Other sockets: 64000

Set to 0 to keep the Java/OS defaults (JVoIP will not set the buffer size in this case).
Set to a positive value to set a fix buffer size for all receive sockets. The send socket buffer size will be set to half of the specified value.
You might also set the sockbuffsize_sig and sockbuffsize_rtp parameters to explicitly specify the buffer size for the signaling / rtp sockets.

jittersize

(int)
Although the jitter size is calculated dynamically, you can modify its behavior with this setting.
0=no jitter,1=extra small,2=small,3=normal,4=big,5=extra big,6=max
Default is 3

Increase the jittersize if audio is breaking up or decrease to minimize delay. The default 3 is a good compromise between quality and latency.

maxjitterpackets

(int)
You can limit the jitter buffer size with this setting.
This is an hard limitation for the jitter algorithm and usually should not be set.
With the jittersize left as default (3) the maximum buffered packet count is limited to around 8, so you might set this parameter to a lower value.
One packet means a received udp packet which might contain one or more audio frame.
For example when using G.729 the typical media stream are with 2 frames/packet. Each frame is 10 msec length.
A jitter limitation of 5 would mean maximum 100 msec to be cached. (while the default setting would allow 8 packet which means 160 msec)
Default value is 70 (which basically means no limitation)

allowspeedup

(int)
Specify whether to enable audio playback speedup on high queue size.
(Instead of dropping packets, it might attempt to speed up the playback rate for a short time until queue size drops below threshold)
Possible values:
-1: Auto Guess (default)
0: No (might drop packets instead of speedup)
1: Yes (might speedup instead of packet drop)

In VoIP it is very common that the call setup/audio device open might take some time (10-400 milliseconds in case of JVoIP, depending on mediaench/device capabilities/audio hardware/driver).
Most SIP clients will simply drop audio packets arrived during this time (when their jitter buffer or other queue is full) which results in a small audio loss (it can be heard sometimes, not at other times).
JVoIP might just speedup the playback for a while to catch-up. This might result in a little audio distortion but will prevent audio loss in such situation.
The same algorithm is applied for burst receive or when the peer is sending more packets then expected (clock/synchronization issues).

allowfirstdrop

(int)
Specify whether to enable drop of RTP packets on media start.
Sometime the other end might start sending RTP before the media is opened at JVoIP side resulting in accumulation of packets.
Dropping the first few packets will make the playback of the rest more smooth, but might result in some loss of audible media (usually up to 100 milliseconds)
Possible values: -1: Auto Guess (default), 0: No, 1: Yes

aqtest

(int)
Audio quality test.
Set to 1 for server/voice quality tests. More details in the FAQ.

loglevel

(int)
Tracing level. Values from 0 to 6.
If you set it to more than 3, then a log window will appear and also will write the logs to a file (if file write permissions are enabled on the client side).
With level 0, the VoIP SDK will not even display important even notifications for the user. Don't use this level if possible.
Loglevel 5 means a full log including SIP signaling. Higher log levels should be avoided in production as it can slow down the application.
Text logs are sent to the following outputs:

-status display (only level 1 –these are the most important events that needs to be displayed also for the user)
-log window (if loglevel is higher than 3 then a log window will appear automatically. Copy the logs with Ctrl+A,Ctrl+C,Ctrl+V)
-file if loglevel is higher than 3 (\mwphonedata\webphonelog.dat near the app or in the java user home directory which depends on the OS/java/browser used)
-java console (if the logtoconsole parameter is set to true)

Default loglevel for demo/trial builds: 5
Default loglevel for licensed builds: 1

More details can be found here.

logtoconsole

(int)
Whether to send tracing to the java console (System.out.print or often referred as STDOUT).
Possible values:

- 0: no (disable log to console)
- 1: auto (depending on loglevel)
- 2: always (always log to stdout)

Default is 1.

logview

(int)
Specify if to show a log window
Possible values:

- 0: never (don't start the log windows at startup and don't collect any logs)

- 1: collect initial logs to be displayed on demand if the loglevel is higher then 2 and not headless
- 2: yes (start the log windows at startup if the loglevel is higher then 2 and not headless)
- 3: yes always and always collect logs (even if the log window is closed)

Default value is 1.

Old parameter name was "canopenlogview" (still works)

canlogtofile

(int)
Specify if JVoIP should write the logs to file.
Possible values:

- 0: never
- 1: if the loglevel is higher then 2 and not headless
- 2: always

Default value is 1.

With loglevel set to 2 or more, the logs are written also to a local file by default.
To disable the log files, set the canlogtofile parameter to "0" (or set the "loglevel" to 1 and keep the canlogtofile default at 1).

The logs are usually stored at \mwphonedata\webphonelog.dat or you can specify the path with the logpath parameter. The exact path can be queried with the API_GetLogPath function.
The log file is recreated with every startup and the old log file is renamed to previous_webphonelog.dat (so you will have max 2 log files: the current and the previous).

logpath

(string)
You might specify the log file path. Environments variables enabled. It might be useful in corporate environments with centralized logging.
Default is empty (which means mwphonedata subfolder or if no write access then in user, tmp or appdata folder, usually at \mwphonedata\webphonelog.dat).
The path can be queried with the API_GetLogPath().

deloldlogs

(boolean)
Specify whether you wish to delete the old log files.
0: no (new logs will be appended)
1: yes (the previous log file will be kept, older files will be deleted)
3: delete old log
Default: 1

With the default settings (1) the log file is recreated with every startup and the old log file is renamed to previous_webphonelog.dat (so you will have max 2 log files: the current and the previous).
This setting cannot be changed at runtime.

capabilityrequest

(boolean)
If set to true then will send a capability request (OPTIONS) message to the SIP server on startup. The serveraddress parameter must be set correctly for this to work. This method is useful to release the security restrictions when using the VoIP SDK with the API and also to open the NAT devices.
This setting cannot be changed at runtime (used only at startup).
Default value is false.

keepaliveival

(int)
NAT keep-alive interval in milliseconds which is usually sent from register endpoints.
Possible values:

- -1: auto (this will default usually to 28 seconds on UDP and 600 seconds on TCP, but can be influenced by many factors)
- 0: disable
- 1-3000: invalid
- 3000 ore more: milliseconds to send keep-alive packets.

Values below 20000 (20 sec) should not be used.
Default value is -1.

recaudiobuffers

(int)
Number of buffers used for audio recording.
Default is 7.

recaudiomode

(int)
Audio recording mode. 0 means default; 1 means event based; 2 means device poll.
Default is 0.

useencryption

(boolean)
Set to true for encrypted communication (both media and signaling)
This is for tunneling with mizu servers and not about the standard TLS/SRTP.
Default is false.

maxlines

(int)
Maximum active port number from 1 to 4. When set to 1, the multiline functionality will be disabled for the built-in user interface.
If you would like to reject all incoming calls if JVoIP is already in a call, then use the “rejectonbusy” parameter instead of setting the maxlines to 1.
Default value is 4.

JVoIP can be use with unlimited simultaneous calls or up to your hardware resources.
The maxlines setting is relevant only if you are using the built-in GUI, otherwise the maxlines will be automatically doubled until it will reach the maxlineex limit, which is 512, by default. This is plenty for normal usage as a SIP endpoint and the limitation is set only to avoid too much CPU/RAM usage under unusual circumstances such as wrong code or DoS attack. Otherwise, you might further increase the maxlineex limit after your needs (for example if you are using JVoIP as a server or gateway and need more than 512 simultaneous calls).

For more details, see this FAQ point.

httpsessiontimeout

(int)
Maximum session time in minutes.
Used only when the VoIP SDK is used as an applet and controlled from java script to avoid situations when the java applet is still running but the user http session is already expired.
You must call the API_HTTPKeepAlive() periodically (for example in every 20 minute) to avoid the timer expiry.
Default value is 60 minute.

singleinstance

(boolean)
If set to true, it will allow only a single JVoIP app instance.
More exactly it will always allow to run the last one, killing the previous once if any.
Default is false

destroymode

(int)
Controls how JVoIP will cleanup itself
Possible values:
-1: auto (full from API_Exit, otherwise weak)
0: weak (static globals will remains)
1: always full
Default is -1

exitmethod

(int)
Controls how the java application or library will cleanup and exit.
Possible values:
-1: auto (default; currently it is the same as 2)
0: do nothing
1: just finish
2: finish and cleanup
3: open exiturl in _self
4: open exiturl in _top
5: call system exit
6: open url and call exit

systemexit

(int)
Specify if JVoIP can call the System.exit() method to quit the JVM when stopped.
0: never
1: auto (Yes if running as a standalone app from command line. No if used as a library)
2: Yes
Default is: 1

fastexit

(int)
Set cleanup speed and wait times.
0: slow but will more care about un-registration and cleanup
1: fast
2: very fast with no unregister
Default is: 1

exiturl

(string)
Specify the URL loaded on exit if the exitmethod is 3,4 or 6.
Default is: https://www.mizu-voip.com/F/webphoneexit.htm (an empty page)

wpapilistenip

(string)
Listen IP address (bind IP) for the built-in API.
By default, the built-in API will listen on the localhost loopback address.
In case if you wish to access the API from a remote machine then you might set it to "all" (to listen on all IP addresses) or to a exact IP address.
Possible values:

- Empty: localhost (default)
- IP address (a valid local IP address on the machine)
- "all": will not bind to any IP
- "0.0.0.0": wildcard address (an IP address chosen by the kernel)

More details here.

wpapilistenport

(int)
API listen port for TCP/HTTP.
By default might be set to 19422, but you should set it explicitly to be sure.

More details here.

wpapiudplistenport

(int)
API listen port for UDP.

By default might be set to 19422, but you should set it explicitly to be sure.
More details here.

wpapiconnectport

(int)
Messages will be sent to this port if UDP socket is used.
Default is: 19421
This will be automatically set to the port from where the webphone received API requests.

More details here.

canuseudpnotifications

(int)
Specify if JVoIP should send old style outband notifications via UDP to 127.0.0.1:wpapiconnectport. This is an old deprecated method.
Possible values:
0: no
1: only if otherwise can't be sent
2: yes auto
other: must (value treated as local port number)

Default is 1.

More details here.

secondarystatus

(int)
Specify if STATUS notifications should be sent also from extra endpoints.
Possible values:
0: no (don't send status from secondary endpoints)
1: yes (send status also from secondary endpoints)
Default is 0.

webphonetojs

(string)
Java script function to be called for the notifications.
Default value is "webphonetojs"

More details here.

jsfunctionpath

(string)
Only if used as an applet.
If your webphonetojs is embedded in other html elements, then you can give the path here. Example: document,externform,innerform2.
By default it is an empty string. This means that Webphonetojs must be placed on the top level (after <body> for example)

More details here.

events

(int)
Defines the level of notifications:
0: no notifications
1: status and cdr
2: important events
3: all logs including SIP signaling messages (depending also on loglevel).
Default is 2.

Old parameter name was javascriptevents, which is deprecated now (but still usable)

stats

(int)
Set to a value in seconds if you wish to receive extended periodic statistics for each line (STATUS notifications).
Default is 0 (no periodic statistics per line; but the global state might still be repeated time to time)

Old parameter name was jsscripstats, which is deprecated now (but still usable)

notificationssingleton

(int)
Specify if to reuse the SIPNotification objects.
-1: auto
0: separate SIPNotification object for all function calls
1: one SIPNotification object to be reused
Default is -1 (defaults to 1)

If set to 1, you will not be able to reuse the SIPNotification (e) object later in your code as it will be always overwritten with the latest.

notificationeventthread

(int)
Notification event threading.
-1: auto
0: from the main thread only (thread safe but slower with around 20 milliseconds delay)
1: from the caller thread (faster but not thread safe. the user must synchronize; don't access your GUI directly from the notification events)
Default: -1 (defaults to 0)

notificationevents

(int)
Notification events vs string send vs string polling.
To get the event notifications from the voip SDK to your application, you can use one of the following methods:

- Notification events using the API_SetNotificationListener to subscribe (to receive SIPNotification objects)
- API_GetNotificationStrings (to blocking receive notification strings in a separate thread)
- API_PollNotificationStrings (polling for notification strings)
- webhonetojs (JavaScript only)
- socket (notification strings sent via UDP or TCP)

Specify which method to use with this parameter. Possible values:
-1: auto/on demand
0: don't use polling (for socket or webphonejs)
1: use polling
2: use polling or socket or webphonejs
3: use polling only (webhonetojs and socket will not work)
4: use SIPNotification event objects only
Default is -1

If set to -1 (default), then

- will turn to 0 on first socket receive or notification events
- will turn to 3 on first API_PollNotificationStrings
- will turn to 4 on first API_SetNotificationListener function call

We recommend to use the newly implemented SIPNotification events (API_SetNotificationListener). The other methods are deprecated, but will remain supported as-is for backward compatibility.

Note: the old parameter name was polling or jsscriptpoll which are still supported

Appearance Parameters

The JVoIP has a simple built-in user interface, which can be convenient if you launch it as a standalone application.
Most of these are ignored if you use JVoIP as a library or launch as a command line application.
In case if you wish to use JVoIP on headless linux systems (without GUI/X-Server/X-Window/VNC installed), ask for the headless version, even if you are not using the JVoIP built-in user interface (the default build can't start on systems without X-Windows due to internal dependencies).

These settings below can be used to control how JVoIP user interface (if any) will look like.
For more customization, you can write your own user interface and use the API to control the Java SIP client library after your needs.
Some of these parameters will take effect only if set at startup (for example from command line) and ignored if set at runtime (via the API).

applet_size_width, applet_size_height

(int)
Should be used only if you use VoIP SDK as an applet embedded in a website.
The size of the space occupied by JVoIP can vary depending on the other parameters.
-if the compact parameter is set to false, than you should set applet_size_width to 300 and applet_size_height to 330.
-If the compact parameter is set to true, than you should set applet_size_width to 240 and applet_size_height to 50.
You can run JVoIP in hidden mode, when all parameters are passed from server side scripts. In this way you can set applet_size_width and applet_size_height to 1.

compact

(boolean)
False: JVoIP will be shown in its full size with username, password input box and dial pad
True: JVoIP will have only a Hangup/Call button and a call status indicator. In this mode the username, password and callto parameters are already set from parameters, so when JVoIP is launched it immediately starts dialing the requested number.
Default value is false.

Usually when this parameter is true, than the “call” is also set true.
Usually when this parameter is false, than the “call” is also set false.

multilinegui

(boolean)
Multiple lines enable the user interface to handle more than one call in the same time. (This doesn’t mean multiple server accounts/registrations. If you need to use JVoIP with multiple VoIP servers at the same time, then just launch more instances)
Set to false to hide line buttons. (The phone will still be able to handle multiple calls automatically)
You can restrict the available virtual lines with the “maxlines” parameter.
When set to true, you might also have to set the “hasvolume” to 1 or 2 and the automute or autohold parameters described in this document.
Default is false.

lookandfeel

(string)
Controls the basic design settings. The following values are defined:

- mizu (on request)
- metal
- windows
- mac
- motif
- platform
- system

Default value is null (system specific design is loaded)

colors

(int)
With these parameters you can customize the colors on the JVoIP basic user interface.
Default value is empty.

The following parameters are defined:

- boxbgcolor
- color_background
- color_label_foreground
- color_edit_background
- color_edit_foreground
- color_buton_background
- color_buton_foreground
- color_buton_dial_background

- color_buton_dial_foreground
- color_other_background

There are 3 ways to specify the color parameter:

- integer number: This number represents an opaque sRGB color with the specified combined RGB value consisting of the red component in bits 16-23, the green component in bits 8-15, and the blue component in bits 0-7
- hex number prefixed with #: representation of the color as a 24-bit integer (htmlcolor)
- the name of the color: the following values are defined: black,blue,cyan,darkgray,gray,gren,lightgray,magneta,orange,pink,red, white and yellow

language

(string)
This is usually a two character language code (for example en for English or it for Italian) or for specific accents/countries you can use the long format such as en-US.
The following languages might be included by default with full or partial translations:

0. en: english (default)
1. ru: russian
2. hu: hungarian
3. ro: romanian
4. de: deutsch
5. it: italian
6. es: spanish
7. tr: turkish
8. pt: portugheze
9. ja: japanese
10. fr: French
11. zh: Chineze

The language parameter can be also specified with the language name (eg. language="Spanish") or code (eg. language=6) for some built-in known languages.

If you wish the status messages to be also translated, set the “translatemode” parameter to 0 (0=all,1=auto guess,2=don't translate js api).

You might also set the **locale** parameter to an ISO 639 alpha-2 or alpha-3 language code, or a language subtag up to 8 characters in length. You might also set the country (ISO 3166 alpha-2 country code or UN M.49 numeric-3 area code) like this: locale=en-US

Localization
To add a new language or change/fix/improve existing strings use this website.
If you don’t have the login credentials, ask Mizutech support.
You will be able to add new translations, change any existing strings and generate a new copy for your JVoIP library with the changes.

charset

(string)

Set the character encoding.
Default is empty (will load the local system default).

Common values are UTF-8 and ISO-8859-1.
Default is UTF-8.

More details:
http://docs.oracle.com/javase/7/docs/api/java/nio/charset/Charset.html
http://www.iana.org/assignments/character-sets/character-sets.xhtml
https://docs.oracle.com/javase/8/docs/technotes/guides/intl/encoding.doc.html

If you need more granular setting then you might use the following parameters:

- charset: default encoding
- charset_sip: for SIP signaling
- charset_media: for audio device names (conversion required only on Windows because of the ASCII only wave audio win32 API)
- charset_bytes: for other byte array – string conversions

For example Chinese users might set the locale parameter to "zh-CH" and the charset_media parameter to "gb2312" to better handle native OS API (for example if the JVoIP uses the Win32 wave API, which is ASCII only internally).

hasconnect

(boolean)
Set to false if you don’t need the connect/register button.

hascall

(int)
0: never
1: hangup only
2: always

hasconference

(boolean)
Set to false if you don’t need the conference button and conference features.

hashold

(boolean)
Set to false if you don’t need the hold button

hasmute

(boolean)
Set to false if you don’t need the mute button

hasredial

(boolean)
Set to false if you don’t need the redial button

hasaudio

(boolean)
Set to false if you don’t need the audio button

hasincomingcallpopup

(int)
Specify if to popup an Accept/Reject dialog on incoming calls from the built-in user-interface,
Possible values:
-1: auto (no if API usage guessed)
0: no (no built-in popup. Handle incoming call from your code as you wish)
1: yes
Default: -1
Set to 0 if you don’t need the popup for the incoming calls (the built-in popup with Accept/Reject buttons).
Note: this will not block the incoming calls; will only hide the default user interface –popup dialog- so most probably you will have to replace it with your own user interface or handle the incoming calls automatically from your application. The old parameter name was “hasincomigcall” and it is still valid.

detectlanpeers

(int)
Auto detect other SIP endpoints on the same LAN (broadcast message) such as colleagues at the network place or family members behind the home wifi.
When other user is found, a NEWUSER event notification is triggered.
Possible values:
0: no (will disable also the listener so other endpoints will not be able to detect it)
1: listen only (will be discoverable)
2: yes find others
Default is 2.

displaychat

(int)
Specify if you wish JVoIP to display a chat window.
Usually not needed if you use it as a library in your app and you will implement your own user interface, but you might still set it to 1 if your app is not focusing on chat features but you still wish to enable this feature for the users (so it will work, even if you don't implement any user interface to handle chat sessions).
Possible values:
-1: auto (default)
0: disable (if text messaging is not required (textmessaging set to 0) or a chat user interface might be implemented by your app)
1: enable (might popup the built-in simple chat form on MESSAGE requests)

hasvolume

(int)
0=no volume controls
1=dynamic (default)
2=vertical
3=horizontal (useful if you disable the multiple lines)
Set to false if you don't need the volume controls button

volumeicons

(int)
0=no
1=text (if hasvolume is set to 3)
2=icons (if hasvolume is set to 2)

displaysipusername

(boolean)
Set to true to display the "Extension" edit box.
Default is false.

When sipusername is set (by parameter, user input or api) then it will be used as the sip username and the username field will be used only for authentification. Otherwise the "username" field will be used for both.

displaydisplayname

(boolean)
Set to true to display the "display name" input box.
Default is false.

hideusernamepwdinput

(boolean)
Set to true if you wish to hide the username/password input controls. Default value is false.

hasgui

(boolean)
Set to false if you are not using the java user interface. (When a custom skin is used). This is an optional setting.
Default is true.

FAQ

How to get my own VoIP SDK?

1. Have a look at the description and pricing at the bottom of the homepage:
   https://www.mizu-voip.com/Software/SIPSDK/JavaSIPSDK.aspx

2. Download and try from:
   https://www.mizu-voip.com/Portals/0/Files/JVoIP.zip
3. Contact Mizutech at webphone@mizu-voip.com with the following details
   -your VoIP server(s) address (ip or domain name). This will be hardcoded in your release; otherwise anybody could just download it from your and use as it owns)
   -your company details for the invoice (if you are representing a company)
4. Mizutech support will send your own JVoIP build within one workday on your payment.
   The payment can be made from the pricing grid or other options (paypal, credit card, wire transfer) can be found here: https://www.mizu-voip.com/Company/Payments.aspx

What about support?

All licenses include also a support plan in the cost.
The support is done mostly by email.
Maintenance upgrades are also free as included with your license plan.
Email to webphone@mizu-voip.com with any issue you might have.
Guaranteed supports hours depend on the purchased license plan and are included in the price.
The support period can be also extended if you need more after the included 1-4 years support period expires. (This is optional. There is no need for any support plan to operate your JVoIP).

Deliverables

You will receive the followings once you purchase the Java VoIP SDK:

- JVoIP itself. This is one single file usually named as "JVoIP.jar" and you will receive your own branded (or white label) build without any demo or trial limitations with lifetime license
- software documentation
- invoice (on request or if you haven't received it before or during the payment)
- support on your request according to the license plan

Custom development

Mizutech can provide custom development services if required.
Please contact us at webphone@mizu-voip.com if you have some extra requirements.
Please contact us only with JVoIP related or VoIP specific projects.

Is it working with any VoIP servers?

Yes. The VoIP SKD uses the SIP protocol standard to communicate with VoIP servers and sofswitches. Since most of the VoIP servers are based on the SIP protocol today, JVoIP should work without any issue.
If you have any incompatibility problem, please contact webphone@mizu-voip.com with a problem description and a detailed log (loglevel set to 5). For more tests please send us your VoIP server address with 3 test accounts.

Is it working with any mobile device?

No. The VoIP SDK works only on devices with support for Java Standard Edition. This means almost all PC/laptop OS and a few linux based phone (not Android).

For Android we have a separate VoIP library with the similar API: Android SIP SDK
For other platforms (iPhone, Android, Symbian) please check our mobile softphones: https://www.mizu-voip.com/Software/MobileSoftphones.aspx

Third-party services

This topic discusses if/when JVoIP uses any Mizutech service or if it will contact any Mizutech servers.

JVoIP is a standalone library, connecting to your VoIP server directly (or directly to SIP peers), without any dependency on our services or third-party services. With other words: if all our servers will be switched off tomorrow, you will be still able to continue using the library and it also works fine in private networks or without internet access.

However by default JVoIP might take advantage of some free online services provided by Mizutech or third parties to ease the usage and for some extra features. All of these are for your convenience. JVoIP will not "call to home" and will not send any sensitive information to Mizutech servers. Most of these are used only under special circumstances and none of these are critical for functionality; all of them can be turned off or changed. These services will not add any overhead and are implemented with minimal resource usage, usually running from low-priority background threads. The following services might be used:

- Mizutech license service: demo, trial or free versions are verified against the license service to prevent unauthorized usage. This can be turned off by purchasing a license and your final build will not have any DRM or "call to home" functionality and will continue to work even if the entire mizutech network is down.
  Note: this is not used at all (completely removed) in paid/licensed versions
- STUN server: in some circumstances JVoIP might use a random stun server hosted by Mizutech. This "fast stun" protocol is usually not required for normal functionality so you might just disable it (set the "use_fast_stun" parameter to 0) or set the "stunserver" parameter to your stun server. However these can improve the connectivity if your VoIP server is not NAT friendly so better to leave it as is. Mizu services (non) availability can't alter the usability of your product.
  Note: you can disable this by setting the fast_stun parameter to 0 or configuring your own fast stun server.
- Alternative IP lookup: in some circumstances the SIP stack might try to auto-detect its public IP address from a randomly selected online service such as mnt.mizu-voip.com, checkip.dyndns.org, wtfismyip.com, icanhazip.com, my-ip.heroku.com, checkip.dyndns.org or ipinfo.io. This is just an extra way to detect the correct external IP which might be useful in some rare circumstances (SIP servers with poor NAT support when STUN is unavailable or bogus so at the first request JVoIP might be already able to present its public address). The failure of this service should not affect the SIP stack functionality at all.
  Note: you can disable this by setting the altexternpublicip parameter to 0.
- Network connectivity: if the SIP stack can't connect to your server or network connectivity have been lost, JVoIP might ping some well know addresses such as google.com to see if there is any network connection at all. This "ping" is then used only to clarify the connectivity problem reports and print appropriate logs to ease the troubleshooting (to separate "No network" from other possible issues such as "SIP server is offline")
  If you don't wish to use a random well-known server, then you can set the networkchecksdomain parameter to any fix domain. For example you might set it to baidu.com if you run JVoIP in China.
  Note: these might be used only if your SIP server is public (not checked if on the same LAN) and you can disable it altogether by setting the networkchecks parameter to 0.
- Geolocation: at first start JVoIP might try to detect its location by a lookup via a random free service such as ip-api.com, ipinfo.io or www.geoplugin.net from a low priority background thread. This is just for your convenience so on your server side you can easily store the client location as this information is just sent by X-Country SIP header. By default this is turned off on private networks.
  Note: this feature has nothing to do with the SIP core functionality and it can be disabled by setting the geolocation parameter to 0.
- Resources: In some circumstances JVoIP might try to load some resources from Mizutech web servers if not found in your deployed package: the ring.wav file, the mediaench dll's and ffmpeg if video is enabled. This happens only if these resources can't be found locally. All of these functionalities can be disabled.
  Note: copy the mediaench files near you app to avoid remote download.

  If you need to white-list (or block for some reason) our servers, here is the address list associated with the above services:
  mnt.mizu-voip.com, www.webvoipphone.com, www.mizu-voip.com
  107.175.156.227, 204.12.197.98, 148.251.28.176 / 28 (148.251.28.177 - 148.251.28.190)

Can I get the source code?

The VoIP SDK is a close-source application. The source code is available only for internal usage and for a higher price.

How to register?

SIP registration means connecting to your SIP server and authenticating.
Using SIP servers with registrations are useful for the following reasons:

- SIP server will learn your application address and can route incoming calls to your application (or other sessions, such as chat, presence, voicemail, etc).
- Reaching SIP endpoints behind NAT's: the register and keep-alive sessions have already opened the NAT (created the port binding in your router) so incoming calls, chat and other messages (incoming INVITE, MESSAGE, etc) will reach your endpoint even if it is behind NAT
- If you preconfigured a SIP server then there is no need to pass the full SIP URI anymore for calls. You can just use the peer username or phone number to call it.
- Some servers doesn't allow calls without a successful previous registration

The normal/usual way for endusers to connect to SIP servers is to use SIP registrations. However registration is optional. You might have a use-case where registrations might not be used or you might use JVoIP for peer to peer calls or you might use JVoIP to make calls to SIP devices which doesn't require registrations. Set the register parameter to 0 to disable registrations.

The SIP stack auto-start behavior can be altered with the startsipstack setting or you can use the API_Start function to launch the instance explicitly.
When started, the SIP stack by default (unless you set the register parameter to 0) will automatically register to your SIP server if you configured the SIP account details (serveraddress, username, password and any other parameters that might be required such as the proxyaddress and sipusername).

Otherwise you might disable auto-start (startsipstack) and/or the auto register (register), pass the above parameters dynamically from your code and use the API_Start and/or the API_Register function to initiate the start/connect/register procedure.

Registration success of failure can be obtained from the REGISTER notifications (especially useful if you are using multiple accounts as this notifications are sent separately per account.

The registration state can be also obtained from the STATUS notification strings as described here.
The registered state can be requested with the API_IsRegistered or API_IsRegisteredEx API.

The failure reason can be also obtained (instead of the above STATUS strings) by using the API_GetRegFailReason API.

Check this FAQ point if you are having problems with connect/register.

Multiple account registration

The VoIP SDK is capable to register multiple accounts at the same time. This can be useful to be able to make and receive calls from multiple accounts or multiple servers.

> *Note:*
> *If you need easier control for the separate accounts then you can also just launch JVoIP multiple times (multiple concurrent instances) with different parameters.*
> *Multiple JVoIP/webphone instances per account requires a bit more system resources –some more RAM for each instance- but it might be more convenient to work with if you have to do different things with the accounts (such as different server/transport protocol/different handling).*
> *Using this feature (multi-account in one instance) might be useful if you have simple requirements (handling the accounts in a similar way) or if you need many accounts (to optimize system resources).*

You can configure multiple accounts using the extraregisteraccounts parameter or the API_RegisterEx function passing a string with the accounts fields. Multiple accounts separated by semicolon.
The accounts must be passed as SIP URI's or with the following fields (fields are to be separated by comma):

serveraddress, username, password, registerinterval, proxyaddress, realm, sipusername, displayname, transport

Most of these fields are optional. The mandatory parameters are the username, password (and the serveraddress if that it is different from your main SIP server).

The SIP accounts can be configured by parameters in the following way:

serveraddress, username, password, registerinterval: primary account
serveraddress2, username2, password2, registerinterval2: second account
…
serveraddressN, usernameN, password, registerintervalN: N account

Or via the API_RegisterEx(String accounts) API call where the accounts are passed as string in the following format:

server,usr,pwd,…;server2,usr2,pwd2,…;
(accounts separated by ; and parameters separated by ,)
Example: wobj.API_RegisterEx("myserver1.com,user1,pwd1;myserver2.com,user2,pwd2,120,,username2");
Or by using SIP URI's: wobj.API_RegisterEx("user1:pwd1@myserver1.com; user2:pwd2@myserver2.com");

Or via the "extraregisteraccounts" parameter which have to be set like this:

server,usr,pwd,…;serverN,usrN,pwdN,…;
or by using SIP URI's: usr1:pwd1@server1;usr2:pwd2@server2;
Example: wobj.API_SetParameter("extraregisteraccounts","myserver2.com,user2,pwd2;myserver3.com,user3,pwd3,120,,username3");

Notes:
-These accounts are marked as "secondary" or "extra" accounts by JVoIP. You should register with a "main" account first by using the main parameters and/or with the API_Register() function
-Up to 99 secondary accounts can be used this way
-The ival/registerinterval parameter is optional (default is 3600 which means one hour)
-All other parameters are applied globally for all account (there is no per account profile). The "proxyaddress", if set, will be applied for primary account only
-You can watch for the REGISTER notifications to find out the register state of the separate accounts
-STATUS about registrations are not reported from secondary accounts unless you set the secondarystatus parameter to 1 (but otherwise you will receive call state STATUS notification while in call as normally)
-If you will connect to more then 10 servers via TCP/TLS then you might need to increase the value of the maxsigstreams parameter (default is 10). You might also set the multiplesigstreams parameter to 2.
-You can unregister all accounts at once or individually using the API_Unregister() function
-By default all the accounts might be unregistered with the API_Register(). You might set the autounregisterextra parameter to 0 to prevent this (possible values for the autounregisterextra parameter: -1: guess/ same as 2, 0: no/never, 1: if there was no extra account register meantime, 2: if there was no extra account unregister meantime, 3: if there was no extra account register or unregister meantime, 4: always unregister)
-The registrations will be kept automatically by JVoIP (re-register when needed; you don't need to explicitly register these accounts again)
-All accounts will be automatically unregistered when JVoIP terminates (except if the needunregister parameter is set to false)
-See the Multiple lines section if you wish to perform multiple simultaneous calls on these accounts

How can I make a call?

- Make sure that the "serveraddress" parameter is set correctly (otherwise you will be able to make calls only to direct SIP URI).
- Optionally: Register to the server. This can be done automatically if the "username" and "password" parameters are preset. Alternatively you can register from API (API_Register) or just let the user to fill in the username/password fields and click on the "Connect" button. If JVoIP is registered, then it can already accept incoming calls (it will do it automatically, or you can handle incoming calls from your application, or you can entirely disable incoming calls)
- Now you can make outgoing calls in the following ways:

- Automatically with JVoIP start. For this you will have to preset the username/password/autocall and callto parameter. Then JVoIP will immediately launch the outgoing call when starts (usually with your page load)
- Just let the users to enter a called number and hit the "Call" button (if you are using the built-in GUI)
- or just call the API_Call function (from user button click or from your business logic)

Read through the parameters to find out more call divert settings, such as auto-answer or forward.

How to handle incoming calls?

To be able to receive incoming calls, make sure that you are registered to your SIP server first. Then you can easily test for example by using any third party softphone and make calls to your application SIP username (or extension id or full URI, as required by your server).

In your project all you have to do is to watch for incoming ringing STATUS notifications and from there you can show any user interface as you wish or handle the call from your code.
For example the following status means that there is an incoming call ringing from 2222 on the first line:
STATUS,1,Ringing,2222,1111,2,Katie,[callid]
Example code:

```
class MyNotificationListener extends SIPNotificationListener
{
   public void onStatus( SIPNotification.Status e)
   {
      if(e.getStatus() == SIPNotification.Status.STATUS_CALL_RINGING && e.getEndpointType() == SIPNotification.Status.DIRECTION_IN)
      {
         System.out.println("\tIncoming call on line "+Integer.toString(e.getLine())+" from "+ e.getPeer());
         //webphoneobj.API_Accept(e.getLine()); //auto accepting the incoming call
      }
   }
}
```

An example can be downloaded from here.

Details about the incoming call can be obtained the STATUS notification itself and/or using any of the following functions:

- API_GetLineDetails(-1)
- API_GetSIPMessage(-1, 0, 1)
- API_GetLastRecInvite()
- API_GetCallerID(-1)
- API_GetIncomingDisplay(-1)

Incoming calls can be also automatically answered as described here. Otherwise just use the API_Accept to connect the call (or the API_Reject to disconnect).

JVoIP is capable for automatic line management so if you don't wish to handle lines explicitly in some specific way, then you can ignore all notifications, except those where the line (first parameter) is -1 (the global state).

Check this FAQ point if you are having problems receiving incoming calls.

ERROR and WARNING in the log

If you set JVoIP loglevel higher than 1 than you will receive messages that are useful only for debugging.

- If you receive a HeadlessException related error, then please contact our support for the headless version.
- If you receive a ring.wav related error, please copy the ring.wav file near your JVoIP.jar file.
- If you receive audio device related errors, please check this FAQ point.

Some ERROR and WARNING messages cannot be considered as a fault as they might appear also under normal circumstances and you should not take special attention for these messages. If there are any issue affecting the normal usage, please send the detailed logs to Mizutech support (webphone@mizu-voip.com) in a text file attachment.

JVoIP is not loading/starting

Check if Java is installed on your machine. You can use any JRE/JDK such as Oracle.

On linux use your package manager to install Java. For example on Ubuntu use the following command:

```
sudo apt install default-jdk
```

Make sure that java is on the path or otherwise start the app with full path such as:
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" -jar JVoIP.jar serveraddress=yoursipserver.com username=USER password=PWD loglevel=5

Contact Mizutech if there are some errors and you can't fix it.

If you are using it as an applet and you see a white page or just a java error on your page that usually means wrong parameters. Please inspect your code and if the problem still persist you should check the java console from your OS.
Make sure that java is working correctly: https://www.java.com/en/download/testjava.jsp

Can't connect to SIP server

If the SDK cannot connect or cannot register to your SIP server, you should verify the followings:

- You have set your SIP server address:port correctly
- Make sure that you are using a SIP username/password valid on your SIP server
- Make sure that the startsipstack and the register parameter is set to 1 or 2. Otherwise use the API_Start() and/or API_Register() functions explicitly.
- Make a test from a regular SIP client such as mizu softphone , Zoiper, Linphone or X-Lite from the same device (if these also doesn't work, then there is some fundamental problem on your server not related to our library or your device firewall or network connection is too restrictive)
- Check if some firewall, NAT or router blocks your device or process or the SIP signaling
- Check the logs
- Possible reasons if there are no response for the REGISTER requests:
    - Wrong serveraddress parameter configured. Incorrect domain/IP, port (your server is listening on the standard SIP 5060 port?) or transport protocol (your server is listening on UDP? you can configure this with the transport parameter)
    - SIP server is down, network is down or server malfunction
    - Some firewall between you and your server (do you have some local firewall blocking the java.exe or some external firewall blocking the required port or UDP altogether?)
    - Server is blocking/ignoring these requests for some reason (fail2ban, firewall, etc) or maybe doesn't like something in the REGISTER sent by JVoIP (although everything is done conform the SIP standards)
    - NAT handling issues (if the SIP server has poor NAT handling capabilities and sends the answer back to some incorrect address or if you are on multiple subnets with conflicting routing rules. You might explicitly set it with the localip/favlocalip parameters)
- If there are no any answer for the REGISTER requests, turn on your server logs and look for the followings:
    - The REGISTER reaches your server?
    - Is there any response triggered (or some error)?
    - If answer is triggered, is it sent to the correct address? (it should be sent to the exact same address from where the server received the REGISTER request)
    - The answer packet reach the client PC? You can use Wireshark to see the packets at network level.
- Send us a detailed client side log if still doesn't work with loglevel set to 5 (from the browser console or from softphone skin help menu)

Failed outgoing calls

By default only the PCMU,PCMA, G.729 and the opus wideband codec's are offered on call setup which might not be enabled on your server or peer UA.
You can enable all other codec's (PCMA, GSM, speex narrowband, iLBC and G.729) with the use_xxx parameters set to 2 or 3 (where xxx is the name of the codec: use_pcma=2, usgsm=2, use_speex=2,use_g729=2,use_ilbc=2).

Some servers has problems with codec negotiation (requiring re-invite which is not support by some devices). In these situations you might disable all codec's and enable only one codec which is supported by your server (try to use G.729 if possible. Otherwise PCMU or PCMA is should be supported by all servers).

JVoIP by default might not allow cross-domain calls. For example if you are registered as a@A.com (to A domain) then you will not be able to make direct calls to b@B.com (to B domain). Contact mizutech support to remove this limitation.

Calls are disconnecting

If the calls are disconnecting after a few second, then try to set the "invrecordroute" parameter to "true" and the "setfinalcodec" to 0.
If the calls are disconnecting at around 100 second, then most probably you are using the demo version which has a 100 second call limit.
If the calls are disconnecting at around 3600 second (1 hour) and you are using the java script API then please check the httpsessiontimeout parameter (you need to call the API_HTTPKeepAlive function periodically from a timer)

Avaya systems

If JVoIP can't connect/register/call with Avaya software/hardware, change to UDP (not TCP or TLS) for the link protocol on your Avaya configuration.

need to call the API_HTTPKeepAlive function periodically from a timer)

Twilio

JVoIP can be used both as an endpoint or as a SIP trunk with Twilio.
There is no any extra settings required for this for JVoIP. Just set the serveraddress parameter correctly to your Twilio domain and you might also enable TLS/SRTP encryption with the transport and the mediaencryption parameters.

Twilio SIP related documentation:

- https://www.twilio.com/docs/glossary/what-is-session-initiation-protocol-sip
- https://www.twilio.com/docs/voice/sip
- https://www.twilio.com/docs/voice/api/sip-interface
- https://www.twilio.com/docs/voice/api/sending-sip
- https://www.twilio.com/docs/voice/api/sip-making-calls

Here is a tutorial blog post about Twilio SIP configuration:

- https://www.twilio.com/en-us/blog/registering-sip-phone-twilio-inbound-outbound

It is also possible to create SIP trunk interconnection with Twilio (this might be useful only if you wish to make many simultaneous calls):

- https://www.twilio.com/en-us/sip-trunking

MS Forefront TMG proxy

If you are using the tunneling module from behind MS Forefront TMG proxy server, TMG will not be able to handle streaming well with default settings. For this reason JVoIP will automatically switch off the streaming and will use packet by packet mode when used with the Mizu VoIP Tunneling. However the firewall built into the TMG server might disable this also after a long call. The best way for TMG users is to allow TCP port 443 of configure VoIP access as described below:
http://technet.microsoft.com/en-us/library/dd441021.aspx
http://technet.microsoft.com/en-us/library/ee690384.aspx

Python

The easiest way to work with JVoIP is Java or some language which is close to Java, such as Kotlin. However, JVoIP can be used from any languages which can compile for JVM such as Clojure, Groovy or Python.
Here is a very basic Python sample code:

```
import os
import socket
import time
from threading import Thread

def notifications(sock):
        while(True):
                bufr = sock.recv(4096)
                print(bufr)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 18620))
t = Thread(target=notifications, args=(s,))
t.start()
s.send("API_Start\r\n")
time.sleep(5)
s.send("API_Register,voip.mizu-voip.com,username,password\r\n")
time.sleep(5)
s.send("API_Call(\"1234\")")
```

More details about working from other programming languages can be found here.

Known limitations

Please test with the demo version first before to purchase a license, to make sure that it fulfills your requirements and it works correctly in your environment.
JVoIP has a long list of features and some combination might not work as expected, depending on your environment (OS/hardware/server/network/etc).
The followings are the know limitations:

- Some Linux distributions doesn't have full duplex audio driver for Java. In these circumstances only one audio stream can be used (JVoIP should be able to handle this automatically with default settings)

- Some OS/driver/hardware configurations might not support wideband audio (in these circumstances java voip component will automatically use narrowband only)
- The G.722.1 codec is supported on Windows only
- The full STUN specification is not implemented due to file size and connection time/speed considerations (a light STUN version is used with a built-in list of available servers if needed)
- AEC might not always capable to eliminate (all) echo, depending on server/peers, speaker/headset, audio device/driver and the environment.
- The VoIP SDK is targeting Java SE platforms only. For android you should use the AJVoIP Android SIP library which is very similar to JVoIP. You might also check the cross platform webphone or the mobile softphone's.
- The demo version doesn't run in headless mode like Linux with no X-Server/X-Window installed. However, we do have also a headless mode build available. Just contact us and we will send the headless version on your request.

Programming languages

Supported programming languages

The JVoIP library can be integrated with your application and used by any programming languages which is capable to emit JVM bytecode or runs or Java Virtual Machine.

This includes Java, Kotlin, Clojure, Scala, Groovy, JRuby, Jython and many more.

MAC related issues

New MAC versions requires audio recording permissions which can be enabled by setting the NSMicrophoneUsageDescription key in your Info.plist file as described here. Additionally you might check the audio permission also from your code (authorizationStatusForMediaType/ AVCaptureDevice requestAccessForMediaType:AVMediaTypeAudio) as described here. If still not granted or if you receive silence/blank audio packets then check this and this topic.

In case if you run your app in background or via launchctl (not as a regular GUI application launched by the user) then you might run into audio permission issue in latest mac releases as described also here. In this case you might set your program as Login Item instead as described here.

If the java (JVM or the browser) is crashing under macOS at the start or end of the calls, please set the "cancloseaudioline" parameter to 3. You might also set the "singleaudiostream" to 5.

If JVoIP doesn't load at all on older MAC versions, then you should check this link.

Linux related issues - startup

Trying to launch the JVoIP GUI might thrown exception: AWTError: Assistive Technology not found: org.GNOME.Accessibility.AtkWrapper.
Solution: (re) install full JRE or uncomment the "assistive_technologies=org.GNOME.Accessibility.AtkWrapper" line
in the in /etc/java-XX/accessibility.properties file.

JVoIP doesn't start from headless/console:
The default JVoIP requires X-Server/X-Window/X11 or other GUI support and full Java SE JRE/JDK install (not the headless version).
Solution: ask mizutech for the headless version. For more details see here.

Linux related issues - audio

Linux in general has worse audio handling then Windows and you might encounter various issues depending on your OS/driver/device.
There are a list of workarounds built in JVoIP, but the audio handling might still fails due to the following reasons:

Missing audio libraries / JRE issue:
Some JRE might not include audio libraries.
Solution: switch to Oracle JRE (or other JRE with audio support).
Headless Java installs might not include audio libraries. For example you might get "no icedtea-sound in java.library.path" exceptions.
Solution: use a full JRE (or other JRE with audio support such as Oracle).

Audio doesn't work or can't open or read the audio device / driver issue:
Some linux audio drivers allow only one audio stream to be opened which might cause audio issues in some circumstances.

Solution: change audio driver to oss/alsa/pulse (other than your current one). Other workarounds: Change JVM (OpenJDK, Oracle, others); Change browser if you are using the JS API. See this FAQ for more.
Restart audio: pulseaudio -k && sudo alsa force-reload

JVM audio handling issues:

Make sure that your JVM works correctly with your audio device and driver.
Try record/playback from any other Java app such as the Java Sound Demo (available also from here)
Solution: use other JRE or other version. Change linux audio driver.

Audio reopen problems:

In case if JVoIP can't open the audio device or other apps can't open the audio after JVoIP have been used it, you should set the cancloseaudioline parameter to 1 and/or the singleaudiostream parameter to 4.

cancloseaudioline: 0=default auto,1=yes,2=no,3=never on mac,4=never
singleaudiostream: 0=no,1=auto guess, 2=one playback only,3=one recorder only (default),4=one recorder and one playback only,5=same as 4 but no tests

No full-duplex audio support by the device or driver:

Some linux devices doesn't have full duplex support.
Solution:

You might try to disable the ringtone (set the "playring" parameter to 0 and check if this will solve the problem).
If these doesn't help, you might set the "cancloseaudioline" parameter to 3 and/or the "singleaudiostream" to 5.

Only one audio stream is allowed:

Some linux devices allows access to audio only for one app (one stream) at a time.
Solution: Make sure that no other app use the audio and the audio handle have been released correctly from the previous usage.

Audio quality is very bad / cluttering:

Solution: Increase the jitter buffer value by setting the jittersize parameter to 5 or 6.

Ringtone during call

On some linux systems the ringtone doesn't stop in time (JVoIP keeps ringing after the call is already connected):
Solution: Set the toneplayermode parameter to 3.

Parameters:

You might try to run JVoIP with the following parameters:
audiorecthread=0 audioplaythread=0 audioplaythread2=0 cancloseaudioline=1 singleaudiostream=1
checkedwidebandok=0 checkeduwidebandok=0 toneplayermode=3
sendearlymedia=0 usecommdevice=0 usenativeaudio=0 preinitwinaudio=0 speakerphoneoutput=1

Other audio related issues:

If you are using PulseAudio (default on Ubuntu), check the troubleshooting guide.
For Java audio workarounds on Linux, see this blog post.
Check the call quality issues FAQ point for more details

RTP warnings

With some SIP servers you might see RTP warning in your server logs.

JVoIP will send a few (maximum 10) short UDP packets (\r\n) to open the media path (also the NAT if any).
For this reason you might see the following or similar Asterisk log entries: "WARNING[8860]: res_rtp_asterisk.c:2019 ast_rtp_read: RTP Read too short" or "Unknown RTP Version 1".
These packets are simply dropped by Asterisk which is the expected behavior. This is not a JVoIP or Asterisk error and will not have any negative impact for the calls. You can safely skip this issue.

You might turn this off by the "natopenpackets" parameter (set to 0). You might also set the "keepaliveival" to 0 and modify the "keepaliveival" (all these might have an impact on JVoIP NAT traversal capability)

Media statistics

Media statistics means RTP/RTCP level details, which can be used to analyze call quality or to display a call quality indicator.

- Basic details: are received with the STATUS notifications such as total rtpsent, rtprec,rtploss, rtplosspercet and server statistics if reported. These and more is also received with the RTP LOG notification.
- Quality reports: can be sent as RTPSTAT notifications if enabled by the rtpstat parameter.
  Alternatively it can be requested with the API_RTPStat function call.
- Voice activity detection: described here
- Logs: You can also see more details in the logs such as total rtp statistics reports for calls longer then 7 seconds if the loglevel is at least 3 as EVENT, rtp stat: sent X rec X loss X X% or the RTP notification. If you set the "loglevel" parameter to at least "5" than the important rtp and media related events are also stored in the logs. You might search for "call details" at the log which also includes media related reports after each call.

Voice activity detection

JVoIP has built-in VAD (voice activity detection) algorithm included.
The talking/silence state of both the local user and the remote peer can be reported.
There are two ways to get VAD details: automatic reports (VAD notifications) or using (polling) the API_VAD function.
To enable VAD reports, set the vad parameter to 4.
Once this is set, you can receive the VAD state either by calling the API_VAD function or by the VAD notifications if the vadstat parameter is set to 3 or 4.
For the notifications interval you might also adjust the vadstat_ival parameter after your needs (default is 3000 in milliseconds, which means a VAD report in every 3 seconds).
If the vadstat is set to 3 then it will report VAD state at periodically. If set to 4 then it will report state changes from silence to speaking or inverse more quickly.
To receive VAD state per line, set the vadbyline parameter accordingly: 0: global only (default), 1: global and receiver by line.
Note: VAD is an estimation and the accuracy of the VAD reports is never perfect.

Video calls

JVoIP is capable to initiate and accept SIP video calls.
You might set the video_config parameter to 8 if you always need video calls instead of audio only.
Use the API_Call funcion with the calltype parameter set to 1 to initiate an outgoing video call.
Use the API_Accept function with the calltype parameter set to 1 to accept a video call.

There is a separate SIP video calls documentation, which explains how to handle video calls with JVoIP, including all the video related parameters and API functions:
https://www.mizu-voip.com/Portals/0/Files/JVoIP_Video.pdf

NAT settings

In the SIP protocol the client endpoints have to send their (correct) address in the SIP signaling, however in many situations the client is not able to detect it's correct public IP (or even the correct private local IP). This is a common problem in the SIP protocol which occurs with clients behind NAT devices (behind routers). The clients have to set its IP address in the following SIP headers: contact, via, SDP connect (used for RTP media). A well written VoIP server should be able to easily handle this situation, but a lot of widely used VoIP server fails in correct NAT detection. RTP routing or offload should be also determined based in this factor (servers should be always route the media between 2 nat-ed endpoint and when at least one endpoint is on public IP than the server should offload the media routing). This is just a short description. The actual implementation might be more complicated.
See the RFC 6314 for more details.

You may have to change Java VoIP toolkit configuration according to your SIP server if you have any problems with devices behind NAT (router, firewall).
If your server has NAT support then set the use_fast_stun to 0 and use_rport parameter to 9 and you should not have any problem with the signaling and media for JVoIP behind NAT. If your server doesn't have NAT support then you should set these settings to 2. In this case JVoIP will always try to discover its external network address.

Example configurations:
If your server can work only with public IP sent in the signaling:
-use_rport 2 or 3
-use_fast_stun: 1 or 2 or 3 (2 is recommended)

If your server can work fine with private IP's in signaling (but not when a wrong public IP is sent in signaling):
-use_rport 9
-use_fast_stun: 0
-optionally you can also set the "udpconnect" parameter to 1

Asterisk is well known about its bad default NAT handling. Instead of detecting the client capabilities automatically it relies on pre-configurations. You should set the nat option to yes for all peers.
More details:
http://www.voip-info.org/wiki/view/NAT+and+VOIP
http://www.voip-info.org/wiki/view/Asterisk+sip+nat
http://www.asteriskguru.com/tutorials/sip_nat_oneway_or_no_audio_asterisk.html

With FreeSWITCH you might set the NDLB-force-rport and aggressive-nat-detection values to true in the sip_profiles configuration.
More details here: https://developer.signalwire.com/freeswitch/FreeSWITCH-Explained/Networking/NAT-Traversal_3375417/

In some special circumstances you might also set the bindip (if your server has multiple network interfaces) and the localip (for example if JVoIP is running on a private address and you wish to specify the external address that is communicated with the clients). You might also set the favlocalip or the bindtolocalip parameters if needed.

Ports and firewalls

The JVoIP SIP library by defaults works with all common/usual firewalls such as the Windows built-in firewall, most third-party firewalls and most routers with their default firewall settings.

On “common/usual” we mean packet filters allowing all outbound and blocking inbound only if there is no outbound leg with no explicitly blocking rules and application enabled for application-level firewalls such as the Windows default firewall.

If your firewall is more restrictive, such as allowing only specific ports, then you must enable the ports used by JVoIP on your firewall rules settings.

A typical SIP session requires the following ports:

- Server side (SIP server listens on the following ports, depending on your SIP server configuration):
  - SIP signaling: usually UDP or TCP 5060 or TLS 5061 (5060/5061 are the default SIP ports, but some SIP servers might be configured to use different ports) or as you configured with the serveraddress/proxyaddress parameters.
  - RTP: a random UDP port (usually an RTP port range can be configured by server administrators, such as 30000-40000)
  - RTCP: usually RTP port + 1 if available (for example if RTP port is 31000, then the RTCP will use port 31001)
- Client side (JVoIP will use the following ports):
  - SIP signaling: a stable random port or as specified by the signalingport parameter.
  - RTP: a random UDP port or as specified by the rtpport parameter. For multiple simultaneous calls it will use the next even ports.
  - RTCP: usually RTP port + 1 if available
  - If you are using video, the video RTP port for the calls will be the RTP port + 10
  - JVoIP will detect if a port is already used (by itself or by any other app) and will search for the next available port in that case

Firewall settings:

- In case if your firewall blocks based on external port, then you will have to enable the ports used by your server (usually 5060 and the RTP port range)
- In case if your firewall blocks based on internal port, then you will have to specify the signalingport and rtpport ports explicitly for JVoIP and enabled these ports on your firewall.
  - For the signaling JVoIP might use additional ports if separate connections are required, usually up to 10 ports, so you should enable the signalingport – signalingport + 10 range.
  - For the RTP port range you should allow 2x the maximum simultaneous calls above the rtpport setting. If you are using also video, then allow 4x max calls + 10 ports (RTP audio, RTCP audio, RTP video, RTCP video).

Some ISP’s in some countries (such as UAE, Egypt, etc) attempt to block most VoIP traffic (forcing users to pay higher costs for the national telecom providers) usually using sophisticated deep packet inspections (DPI) to drop or delay packet streams which looks like VoIP. This kind of packet filtering can’t be handled with the standard SIP/RTP protocols even if you use encryption such as TLS/SRTP. In this case we can recommend to setup a VoIP tunnel gateway which will encrypt and obfuscate all VoIP traffic. Support for this encryption/obfuscation is already built into JVoIP.

Performance optimizations

The SDK by default comes with optimal default value but in some circumstances you might need to further optimize for performance.
If you are running the SDK on some ancient device or any device with low-end CPU or low-powered devices (such as Raspberry Pi), you can optimize the performance with the following settings:

- Make sure that no other processes on your device are running with abnormally high CPU usage (remove or optimize other processes that are unnecessarily consuming the CPU)
- Prioritize low computational codecs with the following settings:
  use_pcmu=3 use_pcma=2 use_g729=1 use_gsm=2 use_speex=1 use_speexwb=1 use_speexuwb=0 use_opusnb=1 use_opuswb=1 use_opusuwb=0 use_opusswb=0 use_ilbc=1 use_g7221=0 use_g7221uwb=0
- If you need more than 500 simultaneous calls then set the maxlineex parameter higher then the maximum number of simultaneous calls.
- Increase the sockbuffsize_sig parameter value if you will have lot's of traffic to don't miss incoming SIP requests. Default is 64000. You might set it to 900000 for example.
- If you will have multiple calls from the same Caller-ID, then set the disablesamecall parameter to 0 (otherwise it will ignore new calls from the same caller)
- Decrease the sockbuffsize_rtp parameter value if you have lot's of traffic with audio only (no video) to save memory. Default is 40000. You might set it to 10000 for example.
- Set the cpuspeed parameter to 1000 or less (or some other value between 300 and 2000. Default is 8000.)
- Set the loglevel to 1 (High log levels will slow down the SDK. But don't set it to 0)
- Disable extra audio processing by setting the followings to 0: aec, aec2, denoise, agc, silencesupress
- Disable video if not needed by setting the video_config parameter to 2
- In case if there are only 1 (or 2) CPU cores available for JVoIP, or if you need many concurrent calls, then set the following parameters to 0 to lower the number of threads: audiorecthread, audioplaythread, audioplaythread2
- Increase the audioqueuemaxsize value. Default is 40. Increasing it can help with short CPU spikes and packet bursts but might increase the playback delay
- Set the waitforstun to 0 or to a smaller value in milliseconds (default is 3500) to reduce startup time. This will affect only the delay until the first REGISTER or INVITE request sent and if disabled/lowered then this first session might not have the best possible IP address offer for NAT handling. SIP servers with good NAT handling can deal with it, but it might cause problems STUN is required and your server has poor NAT handling capabilities.
- Might set the syncincomingsignaling parameter to 0 to disable synchronization for SIP receive (default is 1 which means that incoming packets are queued and processed in receive order in the same main signaling thread. If set to 1, then incoming packets are processed in multiple threads as received which is more threading-unsafe)
- Might set the sendearlytrying parameter to 0 to avoid early message preprocessing (in this case JVoIP might send 100 Trying only with a delay when the request is actually fully processed)
- If the sendearlytrying is left at 1/default, then you might set the sendearlytrying_treshold to 0 to start sending early trying from the very beginning (not only when the incoming message queue reach this threshold)
- Additional optimizations:
  - codecframecount: 2 (or even 4; might be incompatible with some bogus servers)
  - voicerecording: 0
  - syncvoicerec: 0
  - vad: 1
  - plc: false
  - rtcp: false
  - aectype: none
  - enablepresence: 0
  - logview: 0
  - voicemail: 0 or 1
  - sendearlyack: 1
  - registerinterval: 300
  - keepaliveival: 50
  - timer: 20 (not recommended)
  - timer2: 20 (not recommended)
  - notificationevents: 4
  - notificationssingleton: 1
  - notificationeventthread: 1
- In case if you are using the SDK on a server with no audio device, you should to set the useaudiodevicerecord and useaudiodeviceplayback parameters to false.
- Optimize your JVM (for example using a low delay GC) or use a more performant alternative JVM (java virtual machine)
- Optimize or change OS audio system and/or drivers (especially on linux)
- Use the headless version instead of the GUI version
- Fine-tune your JVM (max memory) if you wish to run many instances or simultaneous lines/calls. Also increase the -Dsun.net.maxDatagramSockets JVM parameter (for 1000 simultaneous calls, set -Dsun.net.maxDatagramSockets=2100)

It might be possible that one or two of the above already fixes any performance problems on your devices. In that case it is not necessary to change all of these settings. (Change only the relevant settings what you understood).

Server failover/fallback

Multiple SIP servers can be configured to achieve high availability with failover/failback or load balancing.
There are multiple options to configure server failover:

Using SRV DNS records
Configure SRV DNS records for your SIP servers and let the SIP stack choose the server based on your configuration.
In this case you just need to set the JVoIP serveraddress parameter to your domain name.
JVoIP will not only handle the priority and weight correctly (which itself can be used for failover), but if you have multiple servers then it can pick another as a failover server.
This means that first it will try to connect to the "best" server returned from the DNS query and if fails then it will try to connect to another server returned by your SRV DNS.

Multiple A records
JVoIP can also failover if the server or proxy domain resolution returns multiple A records.
If you wish to disable this behavior for some reason, set the checkmultiarecords parameter to 0.

DNS based failover can be disabled by setting the dnscanfailower parameter to 0. Default value is 2 which means at which retry should attempt to failover.

Internal details: with multiple DNS records then SIP stack uses multi-level failover. First it might try in-place with the GetAlternateServerIP method, then it can try on the endpoint level with the "check next srv record" method and then it can try also globally using the "backupserver" method (which is auto set if there are multi DNS results and it was not configured initially)

Using backup SIP server setting
You can also configure a secondary server for JVoIP using the backupserver parameter, especially if you can't or don't wish to use DNS based failover.
If the SIP stack can't connect to your primary server configured with the serveraddress parameter, then it will try the backupserver.

In case if you wish the SIP stack to try the server availability at the startup procedure, then set the autotransportdetect parameter to true. This should be set if you expect frequent failures with your SIP server. If the autotransportdetect is false (default) then JVoIP will try to register to the primary server first (set by the serveraddress parameter) and will failback to the backup server only on register failure.
With other words if you set the autotransportdetect parameter to true, then there is a small additional delay (around 1 second) on startup even if the primary server is online. If the autotransportdetect parameter is false (the default) then there is no any startup delay but the failover will take a bit more if your primary server is offline.

Note:
In addition to these client side methods, you can of course (also) implement server side failover and HA (high availability) as described here (this document is Mizu server specific, but similar methods can be implemented with/in any third-party SIP servers).

I have call quality issues

Call quality is influenced primarily by the followings:

- Codec used to carry the media
- Network conditions (check also your upload packet loss/delay/jittter)
- Hardware: enough CPU power and quality microphone/speaker (try a headset, try on another device)
- (Missing) AEC and denoise

With loglevel 5 you can see RTP details after each call where the "loss" should be below 5%. Search for "call details" in the log for this.

If you have call quality issues then the followings should be verified:

- whether you have good call quality using a third party softphone from the same location (try X-Lite for example). If not, than the problem should be with your server, termination gateway or bandwidth issues.
- make sure that the CPU load is not near 100% when you are doing the tests

- make sure that you have enough bandwidth/QoS for the codec that you are using
- change the codec (disable/enabled codec's with the codec/prefcodec parameter or with the use_xxx parameters where xxx have to be replaced with the codec name)
- deploy the mediaench module (for AEC and denoise). (Or disable it if it is already deployed and you have bad call quality)
- try to disable the audio enhancements. Set the following parameters to 0: aec, aec2, agc, denoise, usecommdevice
- try to change your audio driver if you are using it on Linux (from oss to alsa or pulseaudio or others)
- fine-tune your audio driver if you are using Linux (for example see the PulseAudio troubleshooting guide which is often the Ubuntu default)
- JVoIP logs (check audio and RTP related log entries)
- wireshark network trace (check missing or duplicated packets; playback the RTP stream from wireshark to see if there is any difference)

In case if you hear distorted/fast-pitch playback during the call or especially in the first few seconds, that might be caused by the allowspeedup setting and you might disable it if you wish by setting it to 0.

I have one way audio

1. Review your server NAT related settings (more details here)
2. Set the "setfinalcodec" parameter to 0 (especially if you are using Asterisk or OpenSIPS)
3. Set use_fast_stun, use_fast_ice to 0 and use_rport to 9 (especially if you are using SIP aware routers). If these don't help, set them to 2.
4. If you are using MizuVoIP server, set the RTP routing to "always" for the user(s)
5. If you are using Asterisk, set the "nat" option to "yes" globally or for the extensions
6. Make sure that you have enabled all codec's
7. Make a test call with only one codec enabled (this will solve codec negotiation issues if any)
8. Try the changes from the next section (Audio device cannot be opened)
9. If you still have one way audio, please make a test with any other softphone from the same PC. If that works, then contact our support with a detailed log (set the" loglevel" parameter to 5 for this)

Audio device cannot be opened

If you can't hear audio, and you can see audio related errors in the logs (with the loglevel parameter set to 5), then make sure that your system has a suitable audio device capable for full duplex playback and recording with 8kHz 16 bit mono:
PCM SIGNED 8000.0 Hz (8 kHz) 16 bit mono (2 bytes/frame) in little-endian

In such circumstances you will usually see something like this in the log:
java.lang.IllegalArgumentException: No line matching interface SourceDataLine supporting format …
catch on audiorecorder.Opendefault (5) No line matching interface TargetDataLine supporting format …

If you have multiple sound drivers then make sure that the system default is workable or set the device explicitly from JVoIP (with the "Audio" button from the default user interface or using the "API_AudioDevice" function call from java-script).

There are two levels where audio access can go wrong:

- OS/driver level:
  To make sure that it is a local PC related issue, please try JVoIP also from some other PC.
  If you are using Windows 10, make sure that microphone access is not blocked.
  Another source for this problem can be if your sound device doesn't support full duplex audio (some wrong Linux drivers has this problem). In this case you might try to disable the ringtone (set the "playring" parameter to 0 and check if this will solve the problem).
- Java level:
  It might be possible that native apps can use audio, but problems arise only with Java (JRE).
  You might verify if the problem is with JVoIP or it is a general Java issue by running any third-party Java app capable for audio playback and recording. For example you can try this app.
  You might reinstall Java or use some other JRE/JDK to fix this kind of problems.
- Other suggestions:
  You might also try to disable the wideband codec's (set the use_opuswb, use_opusswb, use_opusuwb, use_g7221, use_g7221uwb, use_speexwb and use_speexuwb parameters to 0).
  If these doesn't help, you might set the "cancloseaudioline" parameter to 3 and/or the "singleaudiostream" to 5.

P2P

How to use JVoIP for peer-to-peer calls?
Registering to a SIP server has various advantages, but JVoIP can be used also for peer to peer calls without using any SIP server, for example to make direct calls between two JVoIP instances or between JVoIP and any other SIP endpoint (such as a third-party softphone on your local LAN).
In this case just set the register parameter to 0 and set the serveraddress parameter to the peer SIP endpoint address (IP:port) or use full SIP URI to make calls such as API_Call(-1, '1111@192.168.1.8:5678').
To make JVoIP reachable for incoming calls, you should set a fix value for the **signalingport**.
For example if your machine IP address is 192.168.1.8 and you have set the signalingport to 5678, then remote peers can reach your JVoIP instance by calling to sip:1111@192.168.1.8:5678.
Set also the startsipstack parameter to 2 so it will auto start.
You might also set the bindip (if your server has multiple network interfaces) and the localip (for example if JVoIP is running on a private address and you wish to specify the external address that is communicated with the clients). You might also set the favlocalip or the bindtolocalip parameters if needed.

In case if you wish to make direct (not via a SIP server) calls to other peers then you do everything the same way as you would work with a server, with the following changes:

- Set the serveraddress parameter to your peer address. Be aware that SIP apps might not use the default SIP port (5060) so you should check the SIP listener port of the peer first and use the same in the serveraddress parameter (which is usually specified as IP:port in this case)
- Set the setserverfromtarget parameter to 2 if you preconfigured a serveraddress but you need direct calls elsewhere
- Set the register parameter to 0 to disable registrations, because register is usually not required in this use-case

Here is a command line example making a direct call from machine A to machine B (with IP 192.168.1.12):

- Machine B (UAS/server endpoint where you will accept the call):
  java -jar JVoIP.jar startsipstack=2 register=0 signalingport=5070 username=5555 loglevel=5
- Machine A (UAC/client endpoint from where you make the call):
  java -jar JVoIP.jar username=4444 serveraddress=192.168.211.128:5070 callto=5555 startsipstack=2 register=0 autocall=true loglevel=5

Using JVoIP on a server

If you wish to use the library on a server, then make sure to use the headless version if you don't have X-Server/X Window installed (X11 or other windowing system).
This usually happens if you wish to run your command line app from terminal or as a daemon on a linux server.
The headless JVoIP is provided by Mizutech for no extra cost, just ask for it.

Note:

- *You can also run the headless version of JVoIP in a Docker container (forward the required ports as described here)*
- *You can partially test the headless mode also from X-Windows by launching it with the -Djava.awt.headless=true VM parameters (for the headless version only)*
- *Usual error message when you are trying to run the normal (full) version on linux without GUI: "java.awt.HeadlessException: No X11 DISPLAY variable was set, but this program performed an operation which requires it"*
- *Some headless JRE might not come with audio libraries. The workaround for this is to install the full JRE (you can still run JVoIP in headless mode)*
  More linux related workarounds can be fond here.

Additional configurations:

- *Set the video_config parameter to 2 if you wish to disable video*
- *See the ports and firewalls and the performance optimizations FAQ points for additional advises that might be useful if you run JVoIP on a server*

In case if you wish to run JVoIP as a server endpoint (as an UAS or as a SIP server/trunk):

- *Set the signalingport parameter to a fix value (such as 5060) so clients can reach it on a fix port as described also in the P2P FAQ point.*
- *You might also set the bindip (if your server has multiple network interfaces) and the localip (for example if JVoIP is running on a private address and you wish to specify the external address that is communicated with the clients). You might also set the favlocalip or the bindtolocalip parameters if needed.*
- *Set the register parameter to 0 (in case if no registration to a SIP server is required)*
- *Set the startsipstack parameter to 2 (or call the API_Start() function)*
- *Set the disablesamecall parameter to 0 in case if you will use the same caller-id for multiple calls*

In case if you don't wish to receive or send audio or your machine doesn't have audio record or playback device:

- *If the server doesn't have an audio device you might need to set the syncvoicerec parameter to 0, useaudiodevicerecord to false and useaudiodeviceplayback to false. (These are useful if you wish to perform media streaming or voice recording only).*

- *You can also set the defsetmuted parameter to 1 if no playback is required so you will not have to make a separate call to API_Mute (1 means suppressing playback to speaker, 2 means suppress recording and 3 will mute both).*
- *You might also set the mediatimeout parameter to 0 (if it is common to have one way audio such as playback only).*
- *You might also modify the sendrtpondisabled parameter after your needs.*
- *If correct RTP address negotiation is not important then then you can also set the use_fast_stun, use_rtpstun and use_fast_ice parameters to 0.*
- *If no audio recording and send is required then you might set the aec parameters to 0.*

Error creating UDP sockets

In some specific circumstances the java voip component might not be able to create more UDP sockets.
In this case you will see one of the followings in the log:

- ERROR, catch on udp bind Unrecognized Windows Sockets error: 0: Cannot bind
- ERROR, catch on udp bind maximum number of DatagramSockets reached

Possible workarounds:

- Make sure that you enable java on your firewall (first time it will ask the user with default settings)
- Verify your firewall (especially if you are using some third party firewall) and virus scanner.
- Disable ipv6 in your OS or set JVoIP to prefer ipv4 by launching java with the following command: -Djava.net.preferIPv4Stack=true
- Allow more sockets for java sip stack with the following JVM option: -Dsun.net.maxDatagramSockets=60

No ringback tone

Copy the ring.wav to your app folder (near the JVoIP.jar).
You can also use your custom ringtone: any wave file encoded as standard 8 kHz 16 bit mono linear PCM files 128 kbits - 15 kb/sec.
In this case you should also set the ringtone parameter to “ring.wav” or to full path.

If this doesn’t help:
Depending on your softswitch configuration, you might not have ringback tone or early media on call connect.
There are few parameters that can be used in this situation:

- set the “changesptoring” parameter to 3
- set the “natopenpackets” parameter to 10
- set the “earlymedia” parameter to 3
- change the use_fast_stun parameter (try with 0 or 2 or 4)
- set the “nostopring” parameter to 1
- set the “ringincall” parameter to 2

One of these should solve the problem.

Echo

This is about the case when the remote party hear itself back (echo).
(In case if the user at the JVoIP side hears the echo, that usually means bad quality echo cancellation at the peer)

To solve the aec issues you need to set the “aec” parameter to 2. (for better quality also set the “denoise” parameter to 1). Make sure that the dll’s (shipped in the mediaench.zip folder) can be downloaded from your server (there are no warnings in the logs regarding aec initialization). You should store the mediaench.dll and mediaenchx64.dll near JVoIP.jar file on your server and enable the dll mime type (or set the “mediaenchext” parameter to “jar” and rename the dlls to jar: mediaench.jar and mediaenchx64.jar).

For maximum echo cancellation you might set the followings:

- aec: 2
- aec2: 2 or 3
- aectype: "software,hardware,fast,volume"
- denoise: 2
- silencesupress: 1

The AEC is never perfect but after average statistics it is capable to eliminate more than 90% of the echo with around 90% success rate.
The success rate depends on many factor such as device, audio driver, engine, room, network, peers, environment, settings.
(This is if a speaker is used. If the user will use a headset than echo is generated only if the headset is broken).

We can't provide technical support for AEC related issues, because AEC is a best effort algorithm and it is known to not work (partially or at all) in some circumstances and there is no any quick fix for these issues except the above mentioned settings.

Chat is not working

Make sure that your softswitch has support for IM and it is enabled. The Java SIP client is using the MESSAGE protocol for this from the SIP SIMPLE protocol suite as described in RFC 3428.
Surprisingly most default Asterisk installations might not have support for this by default. You might use Kamailio for this purpose or any other softswitch (most of them has support for RFC 3428).
If subsequent chat messages are not sent reliably, set the separatechatdiag parameter to 1.
More details here.

Conference is not working

JVoIP can offer conference features even if your server doesn't have conference support, using it's local RTP mixer.
In case if conference doesn't works, check the followings:

- Make sure that your softswitch has support for multiple simultaneous calls for the same user/device.
- Disable all wideband codec to avoid codec renegotiation with peers
  use_speexwb=1 use_speexuwb=1 use_opusnb=1 use_opuswb=1 use_opusuwb=1 use_opusswb=1 use_g7221=1 use_g7221uwb=1 disablewbforpstn=2

JVoIP doesn't receive incoming calls

To be able to receive calls, JVoIP must be registered to your server by clicking on the "Connect" button on the user interface (or in case if you don't display JVoIP GUI than you can use the "register" parameter with supplied username and password)

Once the SIP client is registered, the server should be able to send incoming calls to it.

The other reason can be if your server doesn't handle NAT properly.
Please try to start JVoIP with use_fast_stun parameter set to 0 and if still not works then try it with 2 or 4.

If the calls are still not coming, please send us a log from JVoIP (set JVoIP loglevel parameter to 5) and also from the caller (your server or remote SIP client)

What is the best codec?

In short:
Wideband (16k+ sampling) codec's has a much better quality then narrowband (8k sampling).
The best call quality can be achieved with wideband OPUS, followed by SPEEX. If the peer doesn't have wideband support then G.711 (PCMU/PCMA) is the best quality narrowband codec but these offers minimal compression, thus in practice usually G.729 is preferred since it provides both good quality and good compression ratio.

Details:
There is no such thing as the "best codec". All commonly used codec's present in JVoIP are well tested and suitable for IP calls and the "best codec" depends mainly on the circumstances.
Wideband codec's such as OPUS and SPEEX has a clear audible quality advantage over narrowband codec's and these should be preferred for VoIP to VoIP calls.
PSTN trunk providers often doesn't have wideband codec capabilities and in this case we usually recommend G.729.
If G.729 is not available in your license plan, than the other codecs are also fine (GSM, speex, iLBC)
Otherwise the G711 codec is the best quality narrowband codec. So if bandwidth is not an issue in your network, than you might prefer PCMU or PCMA (both have the same quality).
Between JVoIP users (or other IP to IP calls) you should prefer wideband codec's (this is why you just always leave the speex wideband and ultra wideband with the highest priority if you have calls between your VoIP users. These will be picked for IP to IP calls and simply omitted for IP to PSTN calls)
To calculate the bandwidth needed, you can use this tool. You might also check this blog entry: Codec misunderstandings

What is the default codec priority?

If you doesn't change the codec priorities with the parameters, than the default codec order will be the following (listed in priority order):

1. Opus all (enabled low priority 2)
2. speex wideband (enabled low priority 2)

3. G.729 (enabled low priority 2)
4. PCMU (enabled low priority 2)
5. PCMA (disabled 1)
6. speex ultrawideband (disabled 1)
7. speex narrowband (disabled 1)
8. GSM (disabled 1)
9. iLBC (disabled -not activated 0)
10. G.722.1 (disabled or not activated 0)

This means that to prefer a codec you just have to add one single line for the parameter:

-use_myfaworitecodec=3

This will automatically enable and put your selected codec as the highest priority one.

If you set all codec with the same priority, then the real priority will be the following:

1. Opus
2. Speex wideband
3. G729
4. G711
5. GSM
6. Speex narrowband
7. iLBC (lowest priority)
8. G.722.1

**Wideband and ultra-wideband can be automatically disabled if your sound card doesn't support the increased sample rate.*
**The other endpoint usually will pick up the first codec, or JVoIP will pick-up the first in this list from the list of codec's sent by the other peer*
**GSM, G.722.1 and speex narrowband and ultra-wideband codec's are disabled by default. Set use_g729,use_gsm, use_speex, etc to 2 or 3 to enable them.*
**The available codec's might also depends on your license.*

How to prefer one codec?

Set its priority to 3 and set the priority for all other codes to 2. In this case you preferred codec will be used whenever the other endpoint supports it and other codec's are used only if otherwise the call would fail.
For example the following parameters will set g.729 as the preferred codec and will enable also pcmu and gsm:

```
-use_g729=3
-use_pcmu=2
-use_pcma=1
-use_gsm=2
-use_speex=1
-use_speexwb=1
-use_speexuwb=1
-use_opusnb=1
-use_opuswb=1
-use_opusuwb=1
-use_opusswb=1
-use_ilbc=1
-use_g7221=1
-use_g7221uwb=1
```

For command line:

```
use_g729=3 use_pcmu=2 use_pcma=1 use_gsm=2 use_speex=1 use_speexwb=1 use_speexuwb=1 use_opusnb=1 use_opuswb=1 use_opusuwb=1 use_opusswb=1 use_ilbc=1 use_g7221=1 use_g7221uwb=1
```

Update:
Latest versions has a prefcodec parameter which can be used to set the highest priority codec instead of using the use_xxx settings.
Example: prefcodec=g729.

How to force only one codec?

Enable only one codec by parameters and disable all others. In this case the call might fail if the other end doesn't support the selected codec.
For example the following parameters will force the softphone to use only PCMU (G.711 uLaw):

```
-use_pcmu=3
-use_pcma=0
```

-use_g729=0
-use_gsm=0
-use_speex=0
-use_speexwb=0
-use_speexuwb=0
-use_opusnb=0
-use_opuswb=0
-use_opusuwb=0
-use_opusswb=0
-use_ilbc=0
-use_g7221=0
-use_g7221uwb=0

If you wish to force a wide-band coded, then set also the followings:
disablewbforpstn=0
disablewbonmac=0
alwaysallowlowcodec=0

If you wish to disable call-retry due to codec incompatibilities, set the codecretry parameter to false.

Update:
Latest versions has a codec parameter so you can easily configure one single codec with this setting instead of using the use_xxx settings.
Example: codec=g729.

How to disable redial

On call failure the SIP engine might auto-redial in some circumstances with changed capabilities.
To disable this behavior, set the following parameters:

hasredial=false
redialonfail=0
autoredial=0
allowrecall=0
codecretry=false
callretryonreject=0

Auto-answer

The SDK has full support for auto answer incoming call (automatically connecting incoming calls as they arrive without the need for user interaction).
Auto-answer can be used for various purposes such as intercom, free-hand mode, walkie-talkie, push to talk, call centers, door station or for any other after your business needs.
Auto-answer can be easily implemented multiple ways:

- **API_Accept**
  Just use the API_Accept function to pickup incoming call after your app logic as described here (you also have the possibility to inspect the caller details and decide which call to allow/auto-accept/reject)
- **Auto accept all calls**
  Set the enableautoaccept parameter to 3 to auto-answer all incoming call.
- **Server-side auto answer**
  Auto-answer can be also triggered/initiated by the SIP server.
  The SDK can auto-answer incoming calls if a special SIP header is sent with the incoming invite. To enable this feature, set the enableautoaccept parameter to 2 first.
  Auto-answer can be triggered be any of the standard or non-standard auto-answer headers such as Call-Info, Alert-Info or Auto-Answer.
  The Answer-After delay is also considered if set by your server (if the autoacceptdelay parameter is left with its -1 default value).
  Example SIP headers that can trigger auto-answer:

```
Alert-Info: info=alert-autoanswer
Call-Info: Answer-After=0
Auto-Answer: normal
Answer-Mode: auto
```

  For example if you are using Asterisk, add something like this into your extensions.conf:

```
extern => 100,1,SIPAddHeader(Call-Info:<sip:>\;answer-after=0)
extern => 100,n,Dial(SIP/1111)
```

It is also possible to use any special SIP header for auto answer (such as something proprietary for your server) with the autoacceptheader parameter.

- **Barge-in calls**
  You can specify enable barge-in hidden calls by using a special SIP header set with the bargeinheader parameter.
  This is similar with the above mentioned server-side initiated auto answer, but barge-in calls are hidden calls and they are often used in callcenters by supervisors to listen to JVoIP calls with the purpose of monitoring agents activity.

It is also possible to accept the incoming call after some delay. This can be specified with the autoacceptdelay parameter. (By default it can be loaded from the Answer-After SIP header flag if received from your server with the incoming INVITE.)

Disconnect reasons

You can receive the call disconnect reason with the CDR notification or using the API_GetLastCallDetails or API_GetDiscReasonText function.
There are endless possibilities why a call can fail, the most common reasons are listed below.
The disconnect reasons are reported in the following format: code text. (So you have the text after a space)
Code:

Is a SIP disconnect request or answer code including BYE, CANCEL or any SIP response code above 300.
If no disconnect message were received or sent then the code is -1 or empty/not set.

Text:

The text is one of the followings:

1. local disconnect reasons (listed below)
2. disconnect reason extracted from SIP Warning or Reason headers
3. response text extracted from the first line of SIP responses (textual representation of the response code)
4. textual representation of the disconnect code

The local disconnect reasons can be one of the followings (extra details might be appended and new disconnect texts might be added in the future):

- notaccepted
- User Hung Up
- bye received
- cancel received
- authentication failed
- endpoint destroy
- no response
- call setup timeout
- ring timeout
- media timeout
- endpoint timeout
- tunneling calltime limit
- call max timer expired
- max call time expired
- max speech time expired
- not acked connection expired
- disconnect on transfer
- transferalways
- transfer timeout
- transfer fail
- transfer done
- transfer terminated
- transfer (other)
- refer received
- cannot start media
- not encrypted
- srtp fail
- srtp init fail
- disc resend
- failed media
- forward
- forwardonbusy
- forwardonnoanswer
- forwardalways
- rejectbusy
- rejectonphonebusy
- call rejected by peer
- rejected by the peer
- rejected
- autoreject
- autoignore
- ignored

The SIP disconnect codes (3xx, 4xx, 5xx, 6xx) are described in the SIP RFC.

With the CDR notification (“Discparty”) or the API_GetLastCallDetails (“Disc By“) you will also receive the party which was initiated the disconnect.
This can be one of the followings:

- 0: unknown/not set
- 1: local JVoIP
- 2: remote peer
- 3: undefined/timeout

You can also get the exact disconnect reason / disconnect message from the SIP signaling using the getsipmessage() function.
For example to extract the disconnect reason from the last incoming message, you can use something like this:

```
String sipmsg = API_GetSIPMessage(-1, 0, 0) ;
```

```
String disconnectcode = "";
if(sipmsg.startsWith("BYE ")) disconnectcode = "BYE";
else if(sipmsg.startsWith("CANCEL ")) disconnectcode = "CANCEL";
else
{
        sipmsg = sipmsg.substring(sipmsg.indexOf(' ')+1);
        sipmsg = sipmsg.substring(0, sipmsg.indexOf(' '));
        sipmsg = sipmsg.trim();
        int codenum = Integer.parseInt(sipmsg);
        if(codenum >= 300) disconnectcode = sipmsg;
}
if(disconnectcode.length() > 0) System.out.println("The disconnect code is: "+disconnectcode);
```

Caller ID display

For **outgoing** calls the Caller ID (CLI)/A number display is controlled by the server and the application at the peer side (be it a VoIP softphone or a pstn/mobile phone). Caller-ID means the remote username, extension number or real phone number as sent by your SIP server. The SIP server might or might not forward the remote real caller-id or might replace it to any arbitrary value (such as the CLI number associated to the user, the user extension or auth id).

You can use the following parameters to influence the caller id display at the remote end:

-username: user id or caller id
-sipusername: auth username used for SIP digest authentication
-displayname: sent in the from/contact headers, might be displayed or ignored by the remote peer

Some VoIP server will suppress the CLI if you are calling to pstn and the number is not a valid DID number or JVoIP account doesn't have a valid DID number assigned (You can buy DID numbers from various providers. This is up to your SIP server configuration and has nothing to do with JVoIP).
The CLI is usually suppressed if you set the caller name to "Anonymous" (hide CLI).

You can also send a P-Preferred-Identity, P-Asserted-Identity or Remote-Party-ID header, although these are commonly used by servers/proxies only. These can be configured with the p_referred_identity, p_asserted_identity or remote_party_id parameter. If the parameter is set to – (a single minus character), then the value will be auto calculated by JVoIP. Otherwise, specify it explicitly. The Privacy header can be also configured with the privacy parameter. See the RFC 3325 for more details.
If you have multiple simultaneous calls then all the above parameters can be also be set separately for each call by using the API_SetLineParameter function.
More extra headers (such as the P-Preferred-Identity) can be added with the API_SetSIPHeader function.

For **incoming** calls the Java softphone will use the caller username, name or display name to display the Caller ID. (SIP From, Contact and Identity fields extracted from the incoming INVITE).
You can also use headers such as preferred-identity to control the Caller ID display.
The Caller-ID is received with the STATUS notifications or you can query it with the API_GetCallerID API. After call disconnect, you will also receive the call details with the CDR notification.
You can also use the API_GetIncomingDisplay API, which can return some more details about the caller such as the display name.
Other incoming SIP headers can be obtained by using the API_GetSIPHeader or API_GetSIPMessage function.

See the following example using different colors for the relevant fields:

- username (caller-id) with **blue**
- sipusername (auth-username) with **red**
- displayname (optional) with green

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4brn7wmmo4h
To: Bob <sip:bob@biloxi.com>
From: "Cullen Alice" <sip:alice@atlanta.com>;tag=871822
P-Preferred-Identity: "Cullen Alice" <sip:alice@atlanta.com>
Contact: <sip:alice@pc33.atlanta.com>
Authorization: Digest username="alice",realm="biloxi.com",nonce="xxx",uri="sip:bob@biloxi.com",response="xxx",algorithm=MD5
Call-ID: a82c41f1b65
CSeq: 1 INVITE
Content-Type: application/sdp
```

Content-Length: 142

...sdp content...

You can learn more here and here.

How to implement IM

IM (Instant message/chat / Text Messaging) is well supported by JVoIP. Make sure that your server has support for SIP MESSAGE.
Related parameters: textmessaging, displaychat

Basic IM:

The core chat functionality can be implemented very easily:

- To send a text message just use the API_SendChat function. Example: API_SendChat(-1, "john", "", "Hi!")
- Incoming text messages can be cached by the CHAT notifications. Example: "CHAT,1,john,Hi!"

Usually you will have to maintain a separate thread with each peer (displaying the conversations on a separate page by peer).

Optional features:

Once the simple chat send/receive is working, you might add some extra functionalities:

- Handle delivery success/failure:
  - You might check the CHATREPORT notifications to notify the user if the message was delivered successfully or failed.
- Typing notifications:
  - You might use the API_SendChatIsComposing function to send typing notification to the peer.
  - Also parse the incoming CHATCOMPOSING messages (usually displaying something like "John is typing…".
- Group chat:
  - If you wish to implement group chat, then you should send/accept the group name which is usually the members separated by | . Example: "emma | john | linda"
  - When sending, use the group parameter of the API_SendChat. Example: API_SendChat(-1, "emma", "emma | john | linda ", "Hi")
  - For receiving, check if the message begins with the "GROUP: xy:" substring where xy is the group.
    You can just use the SIPNotification.Chat.getGroup() function.
  - Usually you have to display each group as a separate thread (separate window for each group name). If group name doesn't exists, then use the peername.
- Offline messaging:
  Offline messaging means queuing messages for later deliver when immediate delivery fails.
  Offline messaging is automatically handled by JVoIP unless you set the offlinechat parameter to 0.

Beyond the IM SIP protocol offered by JVoIP, chat is mostly about user interface. Implement a nice GUI with handy features such as threaded messaging, send picture (using the file send API), emoticons and any other features for your users.
See this FAQ point if you run into any issue related to messaging.

How to USSD

IMS USSD (Unstructured Supplementary Service Data) messages are described in the 3GPP TS 24.390 standard.
This service provides the support for UE or network initiated MMI strings including single requests and dialogs.
It is commonly used to send quick feature codes to/from mobile networks used for various purposes such as balance request, callback, MMI supplementary services and other operations.

First of all, make sure to enable USSD by setting the ims3gpp_ussd parameter to 1 or the ims3gpp parameter to 2 as described here.
Use the API_SendUSSD function to send USSD strings.
Received USSD strings can be cached by the USSD notifications.

A simple USSD message exchange looks like this (with the USSD notifications as strings):

1. API function call: API_SendUSSD(-1, "INVITE", "*135#"); //initiate USSD dialog (for example ask for available credit)
2. notification: USSD,1,1,sent //message successfully sent
3. notification: USSD,1,2,Enter PIN code //incoming USSD message (for example network asking for extra info)

4. API function call: API_SendUSSD(1, "INFO", "1234"); //send USSD answer with INFO
5. notification: USSD,1,1,sent //message successfully sent
6. notification: USSD,1,2,Your credit is 5 USD //incoming USSD message (for example final answer received in BYE)
7. API function call: API_Hangup(1); //just for sure in case if somehow the call was not disconnected by the server

How to DTMF

DTMF means Dual-tone multi-frequency signaling and it is commonly used to send touch tones to the server or the other peer. On phone user interface this is usually handled by handling key press or presenting a phone interface to the user who can press the digits to be sent.

Using the SIP protocol, there are multiple ways to send DTMF digits which can be set by the dtmfmode parameter.
If you are using the built-in user interface then you can send DTMF digits by pressing the number buttons during a call and if a dtmf digit is received, then you will see it displayed on the user interface notification area.
If you are using JVoIP as a library then you can send DTMF digits using the API_Dtmf function.
Example: wobj.API_Dtmf(-1, "89"); //send DTMF tone 8 and 9

DTMF digits can be also added to the called number (with API_Call) after comma (,) or semicolon (;).
If you will use comma, then the dtmf digits are sent immediately on call connect.
If you will use semicolon, then the dtmf digits are sent after call connect with a little delay.
Example: wobj.API_Call(-1, "1234,56"); //this will make a call to 1234 and once the call is connected it will send 56 as DTMF digits.

Feedback about the delivery can be obtained by watching for the INFO notifications (you will receive INFO,OK when the message was successfully sent or INFO,ERROR if failed).
On incoming DTMF you will receive a DTMF notifications.

Custom INFO messages

You can send and receive custom SIP INFO messages in the SIP signaling as described in RFC 2976.
Note: If you are looking to send or receive DTMF digits, then you should look at the above FAQ point instead.

To send a custom INFO message, use the API_Info function.

Feedback about the delivery (if necessary) can be obtained by watching for the INFO notifications (you will receive INFO,OK when the message was successfully sent or INFO,ERROR if failed).
INFO notifications are also triggered for incoming SIP INFO messages (INFO,REC).

For other or more general SIP requests, you might use the API_SendSIPMessage function instead.

Call forward

Call-forward is about to divert a not-yet connected incoming call with 301 or 302 disconnect code, redirecting the caller to another target URI. It works like HTTP redirects.
If you are looking to divert an already connected call then you should use call transfer instead.

Calls can be forwarded using the API_Forward() function (to be called for not yet connected incoming calls) or one of the following parameters: callforwardonbusy, callforwardonnoanswer, callforwardalways.

The disconnect code sent can be specified with the callforwardcode parameter. 301 means Moved Permanently. 302 means Moved Temporarily. Default is 302.
Call forward requests (when JVoIP receives 301/302 for INVITE requests) are handled automatically, unless you set the allowcallredirect parameter to 0.
Diversion (RFC 5806) is also handled by default. You can turn it off by setting the diversion parameter to 0. Default is 1/enabled.

Transfer API usage

Call transfer is about transferring a connected call with SIP REFER.
If you are looking to divert a (not yet connected) incoming call then you should use call forward instead.

The transfer mode can be set with the transfertype parameter and the calls can be transferred with the API_Transfer function.
The call transfer is handled with the SIP REFER method as described in RFC 3515, RFC 5589 and RFC 6665.

Terminology:
- Transferee: initial caller
- Transferor: called party who initiates the call transfer (sends the REFER request)
- Transfer-Target: the destination endpoint who ultimately talks with the transferee (the C party in the above example)
- Unattended transfer: immediate blind transfer to the destination
- Attended transfer: a consultation call will be made first to the transfer target and the actual transfer will happen after this call.

With **unattended transfer** using transfertype 1 the transfer will be executed immediately once you call the API_Transfer function (blind transfer; only a SIP REFER is sent).
With **unattended transfer** using transfertype 6 and holdontransfer 1 or 2 the call might be put on hold before transfer and reloaded on transfer fail.
With **attended transfer** (transfertype 5) you can use the following call-flow, supposing that you are working at A side (JVoIP is the transferor) and wish to transfer B (transferee) to C (transfer-target):

1. A call B (outgoing) or B call to A (incoming)
   A speaking with B
2. Call the API_Transfer(-1,C);
   The call between A and B might be put on hold by A if holdontransfer is 1 or 2.
   A will call to C and connect (consultation call; if call fails, then the call between A and C will be un-hold automatically).
   On 180, 183 or 2xx answer (Session Progress, Ringing or OK) the original call between A and B might be put on hold if holdontransfer is -1 or 3.
   A speaking with C.
   If the call fails, the original call between A and B might be reloaded from hold.
3. The actual call transfer will be initiated when A disconnect the call (API_Hangup).
   REFER message will be sent to B (which tells to B to call C. usually by automatically replacing the A-B call with A-C).
4. After transfer events:
   Transfer related notifications (SIP NOTIFY) can be sent between the endpoints once the call transfer is initiated, reporting the transfer state.
   - If transfer fails (B can't call C) the call between A and B will be will be un-hold automatically (if server sends proper notifications)
   - If the transfer succeeds (B called C) the call between A and B will be will be disconnected automatically (if server sends proper notifications)
   - If the server doesn't support NOTIFY, then the call can be un-hold or disconnected from the API (API_Hold(-2,false), API_Hangup(-1))
5. B speaking with C at this point if the transfer was successful, otherwise call between A and B might be reloaded

Related API's: API_Transfer, API_TransferDialog
Related parameters: transfertype, transfwithreplace, allowreplace, replacetype, discontransfer, disconincomingrefer, inversetransfer, transferdelay, holdontransfer, calltransferalways
Note:
In case if attended transfer doesn't work or if you wish to do it in a different way then JVoIP does, you can also perform attended transfer from your code while the JVoIP transfertype parameter is set unattended transfer (1, 6 or 7).
In this case just initiate the consultation call yourself (using the API_Call function) and then use the API_Transfer function (with unattended transfer) whenever you wish the actual call transfer to happen (usually after hangup with the target).

Call transfer should be used only after call connect as most servers doesn't support the REFER method before connected. If you wish to transfer (redirect) the call before call connect, then you should use the API_Forward function instead.

Security and strictness

In VoIP network, security of the servers are much more important than client side security as the servers must be able to handle all kind of requests from any kind of third-party/middleman malicious/hacking/scanning apps/tools/scripts.
Regardless of this, poorly written client apps might leak user's data, might accept spoofed packets or might not be able to leverage the latest SIP security standards/recommendations.
JVoIP is secure by default.
By default JVoIP stores its configuration data only in local files in encrypted format and it follows the relevant SIP standards and RFC's.
The default parameters can be used for production, but there are a few settings, which you might strengthen for additional security and strictness.
The security/strictness related (security sensitive) parameters are listed below. Change only the parameters which is really required for your use-case and you fully understand as some of these will prevent normal usage if your server doesn't have adequate support.

- Use a secure VoIP server (rigorous authentication and call control, DoS and spoofing protection; as an example you can find mizutech VoIP server security related mechanisms here)
- Use strong SIP auth passwords (the password parameter; don't allow endusers to choose weak passwords on your sign-up procedure)
- Use encrypted signaling and media (SIPS) if your server has support for TLS/SRTP: set both the transport and the mediaencryption parameter to 2. More details here.
- Strengthen the tlspolicy parameter if you wish (0: def-auto, 1: allows self-signed or invalid cert, 2: medium with warning if fails but continue, 3: with server cert validation and disconnect if not secure not allowing failbacks, 4: perform also domain validation)
- Set the strictsrtp parameter to 2 or 3 if possible / if compatible with your server/peers (0: lazy, more compatible, more vulnerable, 1: default, 2: strict, may drop the call, 3: disconnect if no full SRTP protection)
- Use VPN for VoIP (this might be useful only for corporate networks and might require additional configuration by the endusers)
- Use VoIP tunneling/encryption (useful also in countries where VoIP is blocked. JVoIP has built-in support for mizutech VoIP tunneling gateway)
- Set the codec's explicitly to match your server/peers capabilities (with the codec/prefcodec parameters or with the use_xxx prameters)
- Set the enable_3pcc parameter to 0 if you don't need 3PCC features

- Set the allowreplace parameter to 0 if you don't wish to accept transfer with replaces
- Set the enablepresence parameter to 0 if you don't need presence features
- Set the video_config parameter to 2 (disable) and the video parameter to 0 if you don't need video calls
- Set the rejectcallto parameter to 1 (or higher) to reject calls to other (non-local JVoIP) users
- Set the rejectcallsfromunknown parameter to 1 to reject calls from unknown sources (not from your SIP server or proxy; this will disable direct/P2P calls)
- Set the enableautoaccept parameter to 0 to avoid any automatic call accept
- Set the allowcallredirect parameter to 0 to disable call forward requests
- Set the earlymedia parameter to 0 to disable any media before call connect
- Set the disabletransferrequests to 1,2 or 100+ to disable REFER requests (1 means ignore, 2 means 497 answer, 100+ means custom answer)
- Disable call transfer or set it explicitly after your needs: transfertype parameter
- Set the autoredial parameter to 0
- Set the redialonfail parameter to 0
- Set the acceptholdchange to 0 (0=don't allow unhold against our willing, 1=on invite/update only, 2=always, also on incoming OK)
- Set the handleexceptions parameter to 0 (otherwise JVoIP will catch global exceptions …maybe thrown by your app)
- Set the changesystemsettings parameter to 0 to avoid any global JVM changes (any calls to System.setProperty)
- Set the acceptsettingsfromsip parameter to 0 to skip any received x-mparam and x-mparamp headers in SIP messages
- Set the transport parameter explicitly (to avoid the default – 1 auto guess)
- Set the canstransportswitch parameter to false to avoid trying other transport protocol on connection failures
- Set the use_fast_stun parameter to 0 (only if your server has good NAT handling capabilities)
- Set the strictssrc parameter to 1 or 2 (1 will lock to SDP IP:port, 2 will lock to with the initial received SSRC; by default JVoIP already will try to stick with the original RTP received from the address announces in SDP or if that is private then to the original SSRC present in first valid RTP packets. Setting the strictssrc parameter to 1 or 2 will prevent any RTP stream changes which might result in no media issues on any SSRC changes due to any reasons)
- Set the allowmizuservices parameter to 2 (this are used only to help NAT handling and should be disabled only for paranoid security configurations if you don't trust Mizutech at all or you have some related legal rules to not send any packets outside of your network)
- Set the altexternpublicip and extchecklocalip parameters to 0 to disable public IP lookup (which might help in NAT handling also in first messages when Via received/rport was not received yet from your server)
- Set the geolocation parameter to 0 (otherwise JVoIP might lookup for local country and will send it to your server in the SIP signaling –just a helper functionality to determine the client location if this could be of any help for your server)
- Set the dtmfmode parameter explicitly
- Set the focusaudio parameter to 0 to disable lowering the volume of other apps
- Disable agc, aec, denoise (set them to 0) if you don't need extra audio processing
- Set the mustconnect parameter to true to not allow calls without earlier successful registration
- Set the mediatimeout to 0 if you don't need call disconnect on RTP failure.
- Set the autocfgsave parameter to 0,-1, -2 to avoid any configuration cache
- Set the loglevel parameter to 1 to disable detailed logs

How to configure SIPS?

JVoIP has full support for secure SIP: SIP signaling overt TLS (Transport Layer Security) to establish secure connections with your SIP servers.
All you need to enable SIPS is to set the transport parameter to 2 (which means TLS).

More details:
To enable full encryption (both for the signaling and media), set both the transport and the mediaencryption parameter to 2 (SRTP).
If you set the transport to TLS, then make sure that you are using the correct serveraddress parameter. SIP servers usually listens for TLS on port 5061, thus the serveraddress should look like yourdomain:5061.
If the SIP port is configured to 5061 and the transport parameter is not set, then JVoIP might try to use TLS by default if not specified otherwise.
If you wish to change the URI to sips (instead of the default sip) then you can do so by setting the sipproto parameter to "sips" (although this is deprecated by the latest SIP standards).
If you need strict TLS certificate validations (not only encryption) then you might set the tlspolicy parameter to 3 (0: def-auto, 1: allows self-signed or invalid cert, 2: medium with warning if fails but continue, 3: with server cert validation and disconnect if not secure not allowing failbacks, 4: perform also domain validation)
The TLS version to use can be set by the tlsversion parameter (by default it will try to negotiate the highest version supported by both JVoIP and the server. Possible values are "TLS", "TLSv1.0", "TLSv1.1", "TLSv1.2" or any other version supported by the JVM).
If your server requires a specific client certificate, you can set the key-store type by the tlsclientcerttype parameter (pkcs12/jks/jceks), set the file path with the tlsclientcertfile parameter and the keystore password with the tlsclientcertpass parameter.
There is also a API_SetSSLContext function if you wish to set your custom SSLContext.

How to implement PTT?

Push to talk can be easily implemented by using the API_Hold function and/or using auto-answer.
If you wish to use hold and somehow call hold is not well supported by your server or the remote peer, then you might use API_Mute instead.
Other related functions and parameters which you might use are the followings:
API_IsMuted, API_IsOnHold, API_HoldChange, automute, autohold, holdtype, muteonhold, defmute, sendrtponmuted
For ED-137 PTT (air traffic management) use the API_ED137PTT function instead.

RTP header extension

With RTP header extension you can send/receive extra 32 bit words with the RTP headers as described in Section 5.3.1 of RFC 3550.
This is a rarely used RTP feature. Make sure that your VoIP server and/or the peers supports this before to try sending any extra RTP header.
Sending:
If you wish to always send the same data, then you can use the rtpextraheader and the rtpextraheader_profile parameters to set it globally.
If you wish to modify it at runtime or to send different data per call, then use the API_RTPHeaderExtension function.

Receiving:
Set the rtpextraheadernotify parameter to 1 if you wish to receive RTPE notifications about the received RTP extra header changes.
Note: if the ed137 parameter is set, then you will receive RTPT notifications (instead of RTPE). See the ED-137 documentation for more details.

Mediaench module

This FAQ point is about how to deploy the mediaench modules.
The media enhancements module contains optional native media processing libraries, which can improve JVoIP media quality (sound/audio device handling/codec). Just copy the mediaench files near to your app or JVoIP.jar file.
This module might be used if you enable one of the following options: aec, denoise, silencesupress, agc.
Download from here: https://www.mizu-voip.com/Portals/0/Files/mediaench.zip
JVoIP will automatically load the proper library (the dll, so or dylib file depending on the OS) at runtime if found. Otherwise, it will failback to Java code.

JVoIP will search the following paths to find and load the required library (in this order):

1. System.loadLibrary (which scans the java.library.path)
2. System.load(def) (which will search app and system folder)
3. System.load(currentpath);
4. Load from lib path
5. load from getCodeBase()
6. load from user.dir
7. download
8. failback to java if not found

The mediaench module can be disabled by setting the enablemediaench parameter to 0 or by turning off all the related features (set the agc, aec, denoise, silencesupress and usenativeaudio parameters to 0)

Notification strings

For maximum flexibility, the JVoIP SIP library implements also other ways to receive the SIP notifications as strings, as JavaScript function or as UDP or TCP sockets.

Handling the notifications as strings are deprecated now (since the SIPNotification events have been introduced), but we keep it maintained as it might be useful in some special circumstances, for example if you interact with the library from a different app or if you wish to integrate JVoIP into some non Java/JVM app.

There are multiple ways to receive the notifications a strings:

- polling with API_PollNotificationStrings()
- blocking call to API_GetNotificationStrings() (from a separate thread)
- webhonetojs (useful only if you use the library as an applet from JavaScript as described here)
- socket (notifications sent via UDP or TCP if you use the library via socket as described here)
- or subscribing to notifications by using the API_SetNotificationListener() and using the SIPNotification.toString() function to convert the notifications to plain strings

It is also possible to convert the strings to SIPNotification objects by using the API_ParseNotification() function

Full documentation and example:

See the following example and documentation for the details about handling the notifications as strings:

- Test/Example: https://www.mizu-voip.com/Portals/0/Files/JVoIPTest_With_Notification_Strings.zip
- Documentation:
  - PDF: https://www.mizu-voip.com/Portals/0/Files/JVoIP_With_Notification_Strings.pdf
  - HTML: https://www.mizu-voip.com/Portals/0/Files/documentation/jvoip_with_notification_strings/index.html
  - WinHelp: https://www.mizu-voip.com/Portals/0/Files/JVoIP_With_Notification_Strings.chm

In the above example and documentation the notifications are always handled as plain strings, not using the SIPNotificationListener with SIPNotification objects.

Here are some more relevant notification strings:

STATUS statustext strings:
The following **statustext** values are defined **for general status (with line set to -1):**

- Starting...

- Idle.
- Ready
- Connecting…
- Securing…
- Register…
- Registering… (or “Register...”)
- Register Failed
- No network
- Server address unreachable
- Not Registered
- Registered (or “Registered.”)
- Unregistered
- Accept
- Starting Call
- Call
- Call Initiated
- Calling…
- Ringing…
- Incoming…
- In Call (xxx sec)
- Hangup
- Call Finished
- Chat (or Messaging)

Note: general status means the “best” status among all lines. For example if one line is speaking, then the general status will be “In Call”.

The following **statustext** values are defined **for individual lines** (line set to a positive value representing the call channel number starting with 1):

- Unknown (you should not receive this)
- Init (voip library started)
- Ready (sip stack started)
- Outband (notify/options/etc. you should skip this)
- **Register** (from register endpoints) (or “Register…” or “Registering...”)
- Unregister
- Subscribe (presence)
- Chat (IM)
- **CallSetup** (one-time event: call begin)
- Setup (call init)
- InProgress (call init)
- Routed (call init)
- Ringing (SIP 180 received or similar)
- **CallConnect** (one-time event: call was just connected)
- InCall (call is connected)
- Muted (connected call in muted status)
- Hold (connected call in audio hold status)
- Speaking (call is connected)
- Midcall (might be received for transfer, conference, etc. you should treat it like the Speaking status)
- **CallDisconnect** (one-time event: call was just disconnected)
- Finishing (call is about to be finished. Disconnect message sent: BYE, CANCEL or 400-600 code)
- Finished (call is finished. ACK or 200 OK was received or timeout)
- Deletable (endpoint is about to be destroyed. You should skip this)
- Error (you should not receive this)

You will usually have to display the call status for the user, and when a call arrives you might have to display an accept/reject button.
For simplified call management, you might check only the global state or you can just check for the one-time events (CallSetup, CallConnect, CallDisconnect). These are sent only for the actual line (1,2,etc) and not as global state (-1).
Not all call statuses might be sent (for example the Finishing state can be often skipped and you will receive only the Finished state after a call disconnect).

STATUS notifications related to global/main account registration success or failure:

- Proceeding notification strings:
  - Register…
  - Registering… (or “Register...”)
  - Register Failed
- Success notification strings:
  - Registered
  - Registered.
- Failure notification strings:
  - Connection lost
  - No network
  - Server address unreachable

- No response from server
- Server lost
- Authentication failed
- Rejected by server
- Register rejected
- Register expired
- Register failed

Parse notifications code example:

Instead of parsing the strings yourself, you can convert them to SIPNotification objects with API_ParseNotification() or even better: receive the notifications directly as SIPNotification objects by using the API_SetNotificationListener().
Notifications handling is all about a bit of string parsing so maybe better if you just implement it from scratch so you will be more familiar with your own logic then the string handling practices in this example code.

You will have to parse the received strings from your code by first splitting them by line, since more than one line can be received at once (separated by CRLF or with ",NEOL \r\n ", each line represents a new notification). The most important notification is the STATUS message which can be used to learn the SIP stack state (global state or per line state).
Notifications might be prefixed with the "WPNOTIFICATION," string (you should remove this before parsing the rest of the line).
The parameters are separated by comma ','. First you have to check the first parameter (until the first comma) to determine the event type. Then you have to check for the other parameters according to the specification below.

Below is a short code snippet demonstrating basic notification parsing.
In this example we assume that we receive the notifications in a function called *ProcessNotifications* (called from API_PollNotificationStrings, API_GetNotificationStrings or from UDP/TCP socket receive).

```
public void ProcessNotifications(String receivednot)
{
    if (receivednot == null || receivednot.length() < 1) return;

    // we can receive multiple notifications at once, so we split them by CRLF or with ",NEOL \r\n" and we end up with a String array of notifications
    String[] notarray = receivednot.split(",NEOL \r\n");
    for (int i = 0; i < notarray.length; i++)
    {
        String notifywordcontent = notarray[i];
        if (notifywordcontent == null || notifywordcontent.length() < 1) continue;
        notifywordcontent = notifywordcontent.trim();
        notifywordcontent = notifywordcontent.replace("WPNOTIFICATION,", "");

        // now we have a single notification in the "notifywordcontent" String variable
        Log.v("JVOIP", "Received Notification: " + notifywordcontent);

        int pos = 0;
        String notifyword1 = ""; // will hold the notification type
        String notifyword2 = ""; // will hold the second most important String in the STATUS notifications, which is the third parameter, right after the "line' parameter

        // First we are checking the first parameter (until the first comma) to determine the event type.
        // Then we will check for the other parameters.
        pos = notifywordcontent.indexOf(",");
        if(pos > 0)
        {
            notifyword1 = notifywordcontent.substring(0, pos).trim();
            notifywordcontent = notifywordcontent.substring(pos+1, notifywordcontent.length()).trim();
        }
        else
        {
            notifyword1 = "EVENT";
        }

        // Notification type, "notifyword1" can have many values, but the most important ones are the STATUS types.

        // After each call, you will receive a CDR (call detail record). We can parse this to get valuable information about the latest call.
        // CDR,line, peername,caller, called,peeraddress,connecttime,duration,discparty,reasontext
        // Example: CDR,1, 112233, 445566, 112233, voip.mizu-voip.com, 5884, 1429, 2, bye received
        if (notifyword1.equals("CDR"))
        {
            String[] cdrParams = notifywordcontent.split(",");
            String line = cdrParams[0];
            String peername = cdrParams[1];
            String caller = cdrParams[2];
            String called = cdrParams[3];
            String peeraddress = cdrParams[4];
```

```
            String connecttime = cdrParams[5];
            String duration = cdrParams[6];
            String discparty = cdrParams[7];
            String reasontext = cdrParams[8];
        }
        // lets parse a few STATUS notifications
        else if(notifyword1.equals("STATUS"))
        {
            //ignore line number. we are not handling it for now
            pos = notifywordcontent.indexOf(",");
            if(pos > 0) notifywordcontent = notifywordcontent.substring(pos+1, notifywordcontent.length()).trim();

            pos = notifywordcontent.indexOf(",");
            if(pos > 0)
            {
                  notifyword2 = notifywordcontent.substring(0, pos).trim();
                  notifywordcontent = notifywordcontent.substring(pos+1, notifywordcontent.length()).trim();
            }
            else
            {
                  notifyword2 = notifywordcontent;
            }

            if (notifyword2.equals("Registered."))
            {
                // means the SDK is successfully registered to the specified VoIP server
            }
            else if (notifyword2.equals("CallSetup"))
            {
                // a call is in the setup stage
            }
            else if (notifyword2.equals("Ringing"))
            {
                // check the other parameters to see if it an incoming call and display an alert for the user
            }
            else if (notifyword2.equals("CallConnect"))
            {
                // call was just connected
            }
            else if (notifyword2.equals("CallDisconnect"))
            {
                // call was just disconnected
            }
            else if (notifyword1.equals("CHAT"))
            {
                // we received an incoming chat message (parse the other parameters to get the sender name and the text to
be displayed)            }
        }
        else if(notifyword1.equals("ERROR"))
        {
            // we received an error notification; at least log it somewhere
            System.out.println("JVOIP", "ERROR," + notifywordcontent);
        }
        else if(notifyword1.equals("WARNING"))
        {
            // we received a warning notification; at least log it somewhere
            System.out.println("JVOIP", "WARNING," + notifywordcontent);
        }
          else if(notifyword1.equals("EVENT"))
        {
            // display important event for the user
            System.out.println(notifywordcontent);
        }
    }
}
```

External API

It is possible to configure JVoIP to call an external API on notifications.
Of course, you can implement such functionality also in your code as you wish, but for simple use cases this feature might be convenient.
This is executed as a HTTP/HTTPS GET or POST request to the URI after your choice (such as a local service or a cloud API).
You can define up to 98 different API's to be called in different circumstances and/or with different parameters.

This feature can be configured with the following settings (the X can be from 1 to 99):

- **externapiX_url**: API URL (For example: http://example.com/path/keywords)
- **externapiX_data**: post data if any. If the data is not set, then it will be a GET request. If the data is set then it will be a POST request.
- **externapiX_event**: notification(s) when the API call will be triggered. Multiple notifications can be separated by comma. "ALL" means all notifications.

- **externapiX_line**: if set, then the notification will be triggered only from this endpoint line. -9 or null means global status and all lines (no line filter), -2 means all lines (not the global status), -1 means global state, 1+ means an individual line
- **externapiX_dir**: if set, then the notification will be triggered only if direction match this value. -9 or null means all directions (no dir filter), 1 means out, 2 means in (useful for STATUS)
- **externapiX_filter**: if set, then the notification will be triggered only if it contains this substring

You can set any parameter to "null" to clear the previous value if any.

The following keywords will be automatically replaced in the url and data:
- **USERNAME**: local username (the main account username if you are using multiple/extra accounts)
- **SERVERADDRESS**: the configured SIP server address
- **DEVICEID**: a device ID calculated from the machine settings (such as computer name, hardware, OS version, language, etc)
- **JGUID**: an auto generated GUID for this JVoIP instance (generated and saved at first launch)
- **JSEQ**: auto increment sequence number
- **ALLPARAMETERS**: notification parameters
- **PARAMETER_key_EOF**: the key parameter value (replace the key substring with a notification parameter name)

Examples:
API request with the call detail records:
externapi1_url: https://example.com/cdrhandler?ALLPARAMETERS
externapi1_event: CDR

API request on IM and USSD requests:
externapi2_url: https://example.com/messages/?ALLPARAMETERS
externapi2_event: CHAT,USSD

API request for all events:
externapi3_url: http://127.0.0.1:8080/all?ALLPARAMETERS
externapi3_event: ALL

API request for all global state change with the username path, passing all the STATUS notification parameters:
externapi4_url: http://127.0.0.1:8080/USERNAME?ALLPARAMETERS
externapi4_event: STATUS
externapi4_line: -1

API request on incoming calls passing the local username, the Caller-ID and the SIP Call-ID:
externapi5_url: http://example.com/?user=USERNAME&caller=PARAMETER_peername_EOF&callid=PARAMETER_callid_EOF
externapi5_event: STATUS
externapi5_line: -2
externapi5_dir: 2
externapi5_filter: CallSetup

Applet

Although Java Applets have been deprecated by modern browsers we still keep Applet compatibility for those who are interested in this use case.
You might use the VoIP SDK as an applet embedded into your webpage and it will run fine on all Java capable browsers.
Copy JVoIP.jar to your webserver near your html's and refer to it from anywhere from your html with the "applet" tag. Set at least the "serveraddress" parameter to the IP or domain name of your VoIP server (Softswitch/IP-PBX/SIP proxy)

```
<applet
 archive  = "JVoIP.jar"
 codebase = "."
 code     = "webphone.webphone.class"
 name     = "JVoIP"
 width    = "300"
 height   = "330"
 hspace   = "0"
 vspace   = "0"
 align    = "middle"
 mayscript = "true"
 scriptable = "true"
 alt="Enable or install java: https://www.java.com/en/download/index.jsp"
>
<param name = "serveraddress" value = "SERVER_ADDRESS">
<param name = "loglevel" value = "1">
<param name = "MAYSCRIPT" value = "true">
<param name = "scriptable" value = "true">
<param name = "pluginspage" value = "https://java.com/download/">
<param name = "permissions" value = "all-permissions">
```

```
<b>You must enable java or install if not already installed from <a href="https://www.java.com/en/download/index.jsp"> here </a> </b>

</applet>
```

Embed to your website as an applet and use it with any java enabled browser.
You can also disable the default java user interface (compact=true, width=1, height=1) and create your own using html/css or any other teqniques.

The same API is available also from java script which you can use to implement a VoIP application on your website. You might choose to do some actions in the background, present a single "call" or callback button or to display a phone interface. The important steps are the followings:

1. write a function named "webphonetojs" to catch all messages from the VoIP SDK as notification strings. Regarding the incoming messages you can display the status of the phone (registered,ringing,in-call,etc), the most important events and alert the user about incoming calls or chat messages.
2. load JVoIP as applet with the webpage using applet tags or with the "toolkit" method with the proper parameters (serveraddress, etc) (the parameters can be preconfigured, but you can also pass them via the API)
3. get a handle for JVoIP (document. applets[0] or check the skins in the demo package for a more complex example)
4. optionally call the API_Register automatically or when the user click on the "Connect" button (by default if you provide a username and a password parameter, the java voip client will register automatically on startup)
5. call the API_Call function when the user clicks on your "Call" or "Dial" button
6. popup a window (or enable an "accept" button) when you receive notifications about incoming calls to your "webphonetojs" function. Then call API_Accept or API_Reject according the user action
7. if you will present a dial pad for the users, then you might call the API_Dtmf function whenever the user presses a button
8. you might put additional buttons or other controls on your interface for the following functions: audio settings, logout, hold, mute, redial, transfer, conference and others

Short example:

```
<SCRIPT LANGUAGE="javascript">
 function webphonetojs(message)
 {
   //this is an optional function if you would like to be notified about JVoIP events
   //this function will be called by JVoIP
   //you will have to parse the message and act accordingly
   alert(message);
 }

 function do_something()
 {
   document.applets[0].functionname();
 }
</SCRIPT>

<input type=button value='JVoIP action' onClick='do_something()'>
```

Example call-flow:

```
var wp = document.applets[0];

wp.API_Register("11.12.13.14", "sipusername", "sippassword");  //connect to the SIP server
wp.API_Call(-1, "+363012345678"); //make a new call
//wait for connect message by checking the message received on webphonetojs function
//notify the user when the call is ringing or connected
wp.API_Mute(-2,true,0);  //can be called when the user press a button
wp.API_Mute(-2,false,0);  //reenable audio when the user press a button
wp.API_Hangup(-2); //disconnect all calls

...also call the wp.API_HTTPKeepAlive in every 5 minute to avoid session timeout (defined by the httpsessiontimeout parameter)
```

Note:
Java applets are deprecated in recent Chrome, Firefox versions and in other modern browsers.
If you wish to use the SIP library as a Java Applet then we recommend one of the followings:

- IE (Internet Explorer for Windows)
- The Pale Moon browser (Windows & Linux)
- Old Firefox/Chrome/Safari versions

More details about using the webphone on a webpage can be found here and here.
See the API and the parameters sections below for the details about the usage.

Native executable

There are many ways to convert JVoIP or your JVoIP based project into a native executable.

Latest Java versions has this ability built-in using the jpackage tool.
Example usage:
jpackage --name "webphone" --type app-image --input . --main-jar webphone.jar --main-class webphone.webphone -d "out"

See the jpackage documentation for more options.

There are also many other tools which might use other approach for converting a jar to a native executable.

Trick: most of the tools include a JRE with the executable or in the resulting installer. You might use an old JRE to strip the size since JVoIP has support also for J2SE 5.0. However in this case your project should also avoid new Java language features.

Parameter

The "parameter" or "parameters" term might refer to various things, depending on the context.

- JVoIP settings: on "parameter" we usually refer to JVoIP settings. This is the most common meaning if not indicated otherwise.
- API function parameters: the name of the API functions parameters
- Notification parameters: parameters received in notification strings (substrings separated by comma)

Line parameter

For simple use-case the line parameter for the API functions almost always can be just set to **-1**.

More details:
The line parameter is part of most of the API functions and means the channel number.
With this parameter you can specify the session for which a function call to be applied (such as a specific call if there are multiple simultaneous calls).
The following values are defined:

- -2: all channels
- -1: the current channel set previously by API_SetLine or by other functions. Usually this will mean the first channel (1) or the current endpoint in call. The current channel is also auto set on incoming/outgoing call if not specified explicitly.
- 0 : undefined (this should not be received/sent for endpoints in call, but might be used for other endpoints such as register endpoints)
- 1: first channel
- 2 : second channel
- etc

Most commonly, you will have to always pass -1 as the channel number. You will have to use other values only if you will present a GUI were the user can select different lines or if you have some special requirement to handle the lines explicitly.
Otherwise, JVoIP can do this automatically allocating new channels when needed.

The line parameter is also part for most of the notifications to inform you which session triggered the notification. It can be requested with the getLine() function of the received SIPNotification event object.
This is usually set to -1 for the global state or 0,1,2,3… for the individual lines. For this reason you might see duplicated notifications: one sent for the global state, another sent for the actual line state.
If you wish to implement something basic, then it is enough if you check the notifications only for the global state and ignore the line states.
If you wish to handle multi-lines explicitly then you should check also the state of the individual lines (you might ignore the notifications about global state in this case).

See the "Multiple lines" FAQ point below if you wish to handle the lines explicitly.

Multiple lines

Multi-line means the capability to handle more than one call at the same time (multiple channels / multiple concurrent calls / multiple simultaneous calls).
See this FAQ point in case if you are interested in multiple SIP account registrations instead.

Multi-lines are handled automatically by default and you can skip this description if you don't wish to explicitly manipulate the different lines (for example allowing the enduser to hold/forward/transfer or do any other action per line separately).
It is important to note that this functionality can be achieved also by just launching multiple instances. This FAQ is about handling multiple lines in a single instance.

The terms such as "line", "channel" and "session" usually means the same thing in this document and it is used to identify or select a SIP call session.

Multiple lines can be managed by the line parameter of the API functions (most functions has a line parameter) and by checking the line parameter of the STATUS, CDR and other notifications (most notifications has a line number).

By default you don't need to do anything to have multi-line functionality as this is managed automatically with each new call on the first “free” line. This means that you just need to pass -1 as the line number with any function call and look only the notifications with the line number set to -1 (the global phone state).

If you wish to manage the lines explicitly, then you should pass the desired line numbers with the API calls and also look for the notifications received from the individual lines (such as the line parameter of the STATUS notification). This is necessary for example if you wish to create a user interface where the users can select/change to a specific line (line selector or line buttons).

Multi-line vs multi-account

Multiple accounts refers to the feature when you might have more than one SIP account (on the same or separate SIP servers).
Multi-lines is described here and refers to the feature when you might need to handle multiple phone channels usually for multiple simultaneous calls or call transfer.
You can also have multi-accounts and multi-lines at the same time (multiple simultaneous calls via multiple accounts).

Multi-line vs Conference

When we refer to “multi-line” we mean the capability to have multiple calls in progress at the same time. This doesn’t necessarily means conference calls. You can initiate multiple calls by just using the *API_Call* API multiple times (to initiate calls to more than one user/phone), so you can talk with remote peers independently (all pears will hear you unless you use hold on some lines, but the peers will not hear each-others).
To turn multiple calls into a conference, you need to use the *API_Conf* API (or use the conference button from the softphone.html). When you have multiple peers in a conference, all peers can hear each-other.

Usage

- Just set the line parameter to -1 for all/most function calls and parse only notifications with line -1 if you don’t wish to deal with multiple lines explicitly. In this case JVoIP will handle all the details and will guess which lines to use.
- However, if you have some special requirements to handle the lines/calls in a different way, then try to be strict with the line numbers: try to always pass the correct line parameter for the API calls and process the notifications from different lines according to your app requirements.
- If you are using the built-in user interface, set the multilinegui parameter to true to enable the line selector buttons

Disable multi-line

You can disable multi-line functionality with the following settings:

- set the multilinegui parameter to false
- set the rejectonbusy setting to true
- set the maxlines setting to 1

Other related parameters are the automute and autohold settings.

Channel settings

Individual lines are initialized with the global parameters and their behavior might change depending on the circumstances and usage.
Some parameters can be also set line by line using the API_SetLineParameter function (avoid using this when possible and use the other API’s instead to influence the behavior of specific individual lines).

API

Most API functions has a line parameter which you can set to specify the line number.
Also there are two specific API to set/get the current active line:

- API_SetLine(line); // Will set the current line. Just set it before other API calls and the next API calls will be applied for the selected line
- API_GetLine(); //Will return the current active line number. This should be the line which you have set previously except after incoming and outgoing calls (will automatically switch the active line to a new free line for these if the current active line is already occupied by a call)

The active line is also switched automatically on new outgoing or incoming calls (to the line where the new call is handled).

Notifications

Check the line parameter (getLine()) for the STATUS and other notifications and update your line state machine accordingly and/or change your user interface accordingly.

Channels

The following line numbers are defined:

- -3: new endpoint (this is accepted only by some API’s such as API_SendSIPMessage)
- -2: all (some API calls can be applied to all lines. For example calling hangup(-2) will disconnect all current calls)
- -1: current line (means the currently selected line or otherwise the “best” line to be used for the respective API)
- 0: undefined or not call endpoints (this should not be received/sent for calls, but might be used for other lines like the register endpoint)
- 1: first channel
- 2: second channel
- …
- N: channel number X

Note: If you use the API_SetLine with -2 and -1, it might be remembered only for a short time; after that the API_GetLine will report the real active line or “best” line.

API usage example:

```
API_Call(1, "1111");  //make a call on the first line
API_Call (2, "2222");  //make a second call on the second line

//setup conference call between all lines
API_Conf(); //interconnect current lines

//put first call on hold
API_Hold (1,true);

//disconnect the second call
API_Hangup(2, true);
```

Multiple domains

In case if you wish to make a call to server A and a second call to server B, then you can use the API_SetLineParameter function to modify the domain (and any other settings) before the calls like this (example code):

```
API_SetParameter(0, "enabledirectcalls", 3); //you might need to enable cross domain calls first
//if your servers requires also a registration before calls, then proceed as described at the Multiple accounts FAQ point or call the API_Register function separately for the accounts

//setup the first call
API_SetLineParameter(0, "serveraddress", "A", 0); //set server address (domain and/or IP:port)
API_SetLineParameter(0, "transport", "0", 0); //if this server requires UDP
API_SetLineParameter(0, "username", "1001", 0); //set user account valid on server A
API_SetLineParameter(0, "password", "xxx", 0); //SIP password
API_Call(-1,"2001"); //initiate the first call to server A

//wait for the first call to connect, then setup the second call:
API_SetLineParameter(0, "serveraddress", "B", 0); //set server address (domain and/or IP:port)
API_SetLineParameter(0, "transport", "1", 0); //if this server requires TCP
API_SetLineParameter(0, "username", "1002", 0); //set user account valid on server B
API_SetLineParameter(0, "sipusername", "1002", 0); //set if your server requires a different auth username
API_SetLineParameter(0, "displayname", "1002", 0); //set/overwrite any other settings
API_SetLineParameter(0, "password", "xxx", 0); //SIP password
API_Call(-1,"2002"); //initiate the second call to server B
```

Note

If your use-case requires multiple simultaneous calls, then you might wish to set the following parameters:

- focusaudio: 1
- aec: 0
- aec2: 0
- agc: 0

Some API might auto-guess the correct line to use if you supply a wrong line. For example if you call API_Hangup(X) and on the line X there is no call, the SIP stack might disconnect the call on another line if any.

You can use the API_LineToCallID and the API_CallIDToLine if you wish to convert between line number and SIP Call-ID.
Use the API_GetLineStatus, API_GetLineStatusText or API_GetLineDetails functions to obtain the status or details about a session.

The maximum number of channels (simultaneous calls) is controlled by the maxlines parameter (defaults to 4 but will auto increment on demand) and by the maxlineex parameter (default is 512). The maxlineex parameter should be set to higher (to double) then the expected maximum number of simultaneous calls.

Instead of handling multiple simultaneous calls with one JVoIP instance (one webphone object) you might launch separate JVoIP instances per account or per call (new webphone objects). The advantage of this approach is better separations of the SIP sessions (lines) for easy management. The disadvantage is that multiple JVoIP instances will consume more hardware resources (CPU/RAM).

For many instances or simultaneous lines, you should use the headless version and you might need to fine-tune your JVM settings/parameters (max allowed memory). See the performance optimizations FAQ point for more details.

Voice call recording

JVoIP is capable to record the conversations and it can store the result in a stereo voice file (wav, mp3, gsm, ogg) with the left side containing the caller voice and the right side containing the called party voice. The recorded voice files can be saved to the user local device or uploaded to your WEB or FTP server.
You can use the parameters and/or the API to active this feature.

If all calls have to be always recorded, then just set the voice recording parameters after your needs:
voicerecording is the most important parameter to be set.
If you set the voicerecording parameter to 2 or 3 then either the voicerecftp_addr or the http_addr parameter should be also set.

Other related parameters which you might change: voicerecfilename, voicerecformat, syncvoicerec, uploadretry.

If you wish to turn on/off the recording at runtime (for example to record only certain calls or only part of calls) then you can use the **API_VoiceRecord** function. In this case you might set the voicerecording parameter to 0 to disable recording for the rest of the calls.

You can also receive notifications about the voice recording progress/success/failure: VREC.

Video call recording

The video can be already streamed as described here and from there you can process it as you wish (save it to file).
A simplified method for video saving will be also added in the upcoming new major release.
Contact us if you are interested in this functionality.

SIPREC

SIPREC is a standard protocol for call recording.
SIP-based Media Recording (SIPREC) is also supported by JVoIP, implementing the Session Recording Protocol (RFC 7866) and related standards (RFC 7865, RFC 6341 and RFC 7245).
You might use this protocol with compatible endpoints (with SIPREC support) for voice recording. Otherwise you can use the above described call recording capabilities instead.

Common terms:

- CS: Communication Session (the "call" between participants)
- RS: Recording Session (the "call" between SRC and SRS)
- SRC: Session Recording Client
- SRS: Session Recording Server

The SRC usually runs the CS and RS at the same time, sending the streams to be recorded to the SRS which will then record (store or forward) the call.
JVoIP can act both as an SRC and an SRS.

Parameters:

- siprec_src: 0: explicitly opt out (send recordpref:off), 1: no (default), 2: enable (yes, negotiable), 3: force (always offer/accept)
- siprec_src_start: 0: start recording on call connect, 1: start recording on media start, 2: start recording on call init, 3: always record
- siprec_srs_uri: set to the SRS URI if we are SRC
- siprec_srs: 0: disable, 1: no (default), 2: enable (negotiable, no by default), 3: enable (negotiable, yes by default), 4: always (always force offer/accept)
- siprec_srs_hide: hidden SRS endpoints. -1: default/auto, 0: no, 1: yes (it might be useful to hide RS endpoints if JVoIP acts also as a normal UA, not dedicated for SRS only)

Usage as SRC:
Configuring JVoIP SRC will allow JVoIP to stream its calls to a remote SRS server to be recorded by the SRS.
Set the siprec_src parameter to 2 or 3 to enable SRC functionality and set the siprec_srs_uri parameter to the SRS URI if it is not your SIP server.
Once set, JVoIP will initiate RS sessions with the SRS sending the in/out media streams to be recorded.
Large messages (including metadata XML) are being sent for the SRC calls. You might set the transport protocol to TCP (transport=1) or TLS (transport=2) if your network don't support high enough MTU for the UDP packets.

Example SRC configuration:
serveraddress=yoursipdomain.com username=USR password=PWD siprec_src=2 siprec_srs_uri=IP:port

Usage as SRS:
Running JVoIP as SRS will turn it to a recording server to auto-accept RC calls from SRC's.
Set the siprec_srs parameter to 2, 3 or 4 to run JVoIP as an SRS server.
On new RS calls you will receive SRS notifications with the details about the CS session in the following format:
SRS,line,session_id,sip_callid,sipsessionid1,userid1,aor1,name1,sipsessionid2,userid2,aor2,name2,codec
The sip_callid might not be sent by all SRC's (not a standard parameter).
You can also use the API and the other notifications to obtain more details about the RS sessions. For example you might use the API_GetSIPMessage function to get the full text of the incoming INVITE, including the metadata XML and obtain more data from there according to RFC 7865.

The received media streams can be saved (stored) to file (including uploading the file to your web or ftp server) or you can stream (forward) it in real-time to your app or to any remote service. The first stream (usually with label:1) is send/stored as the incoming (played) stream for normal call sessions and the second stream (usually with label:2) is send/stored as the outgoing (recorded) stream for normal call sessions.

When running JVoIP as SRS, you might consider the settings mentioned in the Using JVoIP on a server FAQ point.
Example SRC configuration (The first few parameters are the most important. The rest are for optimizations):
siprec_srs=4 voicerecording=1 signalingport=5060 register=0 startsipstack=2 syncvoicerec=0 useaudiodevicerecord=false useaudiodeviceplayback=false hasincomingcallpopup=0 answerwithallcodec=1 mediatimeout=0 rtpkeepaliveival=0 use_fast_stun=0 use_rtpstun=0 use_fast_ice=0 aec=0 aec2=0 agc=0 vad=0 denoise=0 p2prtp=0 p2prtpencryt=0 video_config=2 playring=0 ringincall=0 beeponincoming=0 beeptype=0 systembeeptype=0

VREC notifications might be also sent (except if voicrecording is enabled for non-SRS sessions as in that case the VREC notifications are sent for the voicrecording instead and not for SIPREC).
Debug: search for “siprec”, “set siprec”, “start siprec”, “call details”, ”SIPREC:”, ”VREC” and ”SRS” in the logs.

Limitations:

- A new RS session will be created for each call (continuous RS/persistent recording is not supported). SRC should initiate new RS session separately for each individual call.
- The RS sessions can handle only normal audio calls with two streams (between two parties in/out – recv/send) and the streams should have negotiated the same codec. Call groups are not supported.
- Session updates, renegotiations, SRTP, RTCP and metadata updates are not supported for the RS sessions

How to send a media stream?

The VoIP SDK is capable to playback media (audio and/or video) streams to the remote peer. You just need to use the API_PlaySound function to play a sound file to the remote or the API_StreamSoundBuff/API_StreamSoundStream function if you wish to play from buffer/stream in real time.
Technically the only difference from normal calls is that in this case you will stream audio from a file or from a custom buffer/stream instead of from the microphone/audio input device.

Here are the detailed systematic instructions for this use-case:

1. **Configure JVoIP** with the required parameters/credentials by using the API_SetParameter(s) function and passing the serveraddress, username, password and any other parameters after your needs.
   You might set the useaudiodevicerecord parameter to false if you don’t need recording from the local audio device.
2. JVoIP can **register** automatically on startup if you passed the credentials, or you can set the register parameter to 0 if no registration is required or you might use the API_Register function to register explicitly.
3. To **make a call**, just use the API_Call function.
   Until this we just described the basic usage and for all these you can also find a simple **example code** here or inspect the sample downloadable from here.
4. To **detect call failure** (such as remote peer busy or doesn't respond), watch for the STATUS notifications for call disconnects or for the CDR notifications.
   After each call you will receive a CDR notification (with the duration parameter set to 0 if the call failed) and from there you can redial if you wish by calling the API_Call function again.
5. To **stream sound to the peer**, wait for call connect (STATUS notification with the statustext set to "In Call" on line -1 or "CallConnect" on the specific line) and then call the API_PlaySound function to stream from file.
   Instead of API_PlaySound, you might use the API_StreamSoundBuff or the API_StreamSoundStream function if you want to play from buffer or stream instead.
6. If you wish to stream (also) **video**, then use the API_SendVideoRTP function. See the video guide and the video streaming guide for more details.
7. If you wish to **disconnect the call** once the streaming is ready, then just call the API_Hangup function when you receive the PLAYREADY notification.

In case **if you don't wish to record or playback to audio device** at all (only stream from file or buffer), then set the useaudiodeviceplayback and useaudiodevicerecord parameters to false. Please check the "Using JVoIP on a server" FAQ point for more details about this.

How to get the media stream?

The VoIP SDK is capable to stream the received/sent media to your app (your local application or other application which will process the audio/video data).

You can receive the media stream in two ways:

- **API_GetMedia**()
  Receive the media packets from JVoIP by calling the API_GetMedia function from your app (from a thread).
  Set the sendmedia_mode parameter to 2.
  This is the new recommended way for local audio streaming.
  Code example here.
- **UDP packets**
  Receive the media packets from JVoIP over UDP.
  Set the sendmedia_mode parameter to 1.
  Launch an UDP server socket to listen on any port and set the same port number as the sendmediain_to and/or sendmediaout_to parameter for the SDK.
  Then you will receive the media streams (the audio and/or video packets) from the calls as UDP packets and from there you can process them as you wish.
  The UDP packets can be sent also elsewhere as specified by the sendmedia_to_ip parameter.
  This is the old way for local audio streaming (will remain fully supported).
  Code example here.

“Stream” and “streaming” in this context means sending the RTP or the raw audio packets to a non-VoIP application (such as your app).
It doesn’t meant the usual VoIP RTP streams (streaming RTP to remote VoIP peers or from/to SIP server).

Streaming is useful if you wish to make some processing on the media streams such as custom audio playback, video recording, speech to text, AI speech processing or forward to other API/SDK/cloud service for any further processing. For example real-time translation via the Azure or Google Cloud API, voicebot or ChatGPT integration.

Technically the only difference from normal calls is that in this case the media (the incoming and/or outgoing RTP media streams) will be forwarded to your UDP listener or received by API_GetMedia function calls, instead to be sent to the speaker/audio output device (it can be also streamed to both your listener and the audio device in the same time or your can set the useaudiodeviceplayback to false or mute the call to disable playback on the sound device as described here).

Parameters:

sendmedia_mode

Specify how to get the media streams
-1: auto (will be set to 1 if you set the sendmediain_to or sendmediaout_to parameter; will be set to 2 if you call the API_GetMedia function)
0: disable
1: UDP (received on UDP socket)
2: API (received by the API_GetMedia function)
3: both (this is usually unnecessary/not recommended)

sendmedia_atype

Specify the audio media stream format sent by JVoIP to your app:
0: raw wave format (linear PCM) for narrowband (8 kHz 16 bit mono PCM files at 128 kbits - 15 kb/sec) or 16 kHz for wideband (depending on the codec used for the SIP call with the peer)
1: for RTP format (RTP header + payload with the actual media codec)
2: convert sample rate to raw PCM 8kHz (useful if original stream is in 16kHz format such as Opus or Speex wideband, but you need 8kHz PCM)
3: convert sample rate to raw PCM 16kHz (useful if original stream is in narrowband 8kHz format but you need 16kHz PCM)
4: RTP data only, without RTP header (raw codec format)
5: convert to L16 mono, 16khz, Big Endian (16-bit signed linear PCM)
6: convert to L16 mono, 16khz, Little Endian (16-bit signed linear PCM)

Default is 0.
Note:
This parameter is applied only for audio data. Video is always sent as unmodified RTP packets.
It is recommended to set this to 0 or 1 and force a codec to match your needs (use OPUS, Speex or G.722.1 if you need 16kHz PCM wideband or other codec if you need 8kHz PCM narrowband), however if you need raw audio in other sample rate then you might set it to 2 or 3 for sample rate conversion.
With Google STT we recommend to set the sendmedia_atype to 3 and set the codec encoding: LINEAR16 and sampleRateHertz: 16000 for the Google speech API.

sendmedia_mtype

Specify media type:
-1: default (usually will default to 4)
0: undefined/reserved
1: audio only
2: video only
3: both with media type header byte (the first byte will be set to 1 for audio data or to 2 for video data)
4: both, without media type header byte

sendmedia_dir

Specify which streams to forward.
-1: auto (will be set to 1,2 or 3 depending on the sendmediain_to or sendmediaout_to parameter)
0: undefined/reserved
1: incoming (RTP received to JVoIP from the remote peer)
2: outgoing (RTP sent by JVoIP to remote peer)
3: both with direction header byte (the first byte will be set to 1 for incoming RTP or to 2 for outgoing)
4: both, without media type header byte

sendmedia_line

Specify if packets should have a line number header. Useful in case of you wish to handle multi lines in the same stream (sent to same UDP port).
0: no header (default)
1: add line header string. In this case the packets will begin with the line number, followed by comma, followed by the media payload (binary data).
Example: 2,xxxxx
(You must extract the line number from each packet: get the string until the first comma and convert it to int. The bytes after the comma will be the audio data)
2: add line and SIP Call-ID string header. Example: 3,abc,xxxxx
3: add line header byte. In this case the first byte in the packet will be the line number (0-127). This should be used only if you have less than ~100 simultaneous calls.
4: add line header byte and VAD status byte. The VAD status (second byte) will be 0 if unknown, 1 if silence, 2 if speaking.

sendmedia_marks

Specify if you wish to receive BOF/EOF packets (udp packets containing 3 bytes at the beginning and at the end of streams).
Default is 0. Set to 1 if you wish to have these packets.

sendmedia_conf

Set to 1 if you wish to receive the audio streams in conference calls separately (separate stream for each endpoints).
Default is 0 which means that you will receive a single stream (with conference mixed audio packets)

sendmediain_to

Specify your UDP port where you wish to receive remote media (received from other peer for playback on local speaker)
Default is 0 which means no streaming.
Not required if you use the API_GetMedia (instead of UDP socket)

sendmediaout_to

Specify your UDP port where you wish to receive local media (recorded from microphone, which is sent to the other end)
Default is 0 which means no streaming.
Not required if you use the API_GetMedia (instead of UDP socket)

sendmedia_to_ip

Specify the IP address where the media have to be forwarded (to be used with the above ports)
Default is 127.0.0.1.
Not required if you use the API_GetMedia (instead of UDP socket)

For example if you are interested only in incoming audio in raw PCM 8kHz format, then set the following parameters:

- sendmedia_mtype: 1 //audio only
- sendmedia_dir: 1 //incoming only
- sendmedia_atype: 2 //narrowband

Code examples:

- A simple working Java example using API_GetMedia can be downloaded from here.
- An old style streaming example using UDP packets instead of API calls can be found here.

Note:

- The *sendmedia_mtype*, *sendmedia_dir* and *sendmedia_line* settings might insert additional bytes at the beginning of the media buffer. You should process or ignore them (in the above order).
- In case if you just wish to store the audio and don't need real-time processing, then you should use voicerecording / API_VoiceRecord instead of this streaming as described here.
- If you just need to stream audio to a remote SIP endpoint, then use the API_PlaySound or API_StreamSoundBuff function instead.
- You might force G.711 codec to simplify the audio formats conversion (disable all codec's except PCMU and PCMA).
- If you are interested in video streaming then see the video streaming guide for more details.

Work with audio streams

If you have set audio streaming as it was discussed above, then you have the following possibilities to handle it:

1. **Third party software:**

Use some third-party software which is capable to playback or process the audio packets (such as for playback, transcription or monitoring).
For example ffmpeg or a speech to text library (STT).

2. **Cloud service:**

Use some third-party software which is capable to playback or process the audio packets (such as for playback, transcription or monitoring)
Stream the audio to any cloud service for further processing (for example Google, AWS or Azure speech to text API).
For the Google speech API you can find more details here. Contact us if you need some C# sample code *(ref num 8117459355).*
If you are using JVoIP with C#, then you might check this code.
For AWS you can check this code.

3. **RTP stream:**

If you have a software which is capable to process RTP packets then just set the sendmedia_atype to 1 so the JVoIP will emit RTP packets which can be processed as is by such software.

4. **Receive the audio packets with your app:**

You just need to start and UDP server (preferably in a separate thread, listening on the port that you specified as the sendmediain_to and/or sendmediain_from JVoIP parameter) and you will receive the audio in the UDP packets.

In case if you are using Java and wish to work with streams instead of byte buffers, then just convert the received audio byte buffer to stream.
Example: InputStream is = new ByteArrayInputStream(bytearray); //you can convert to OutputStream or other stream types similar way

Example Java code:

```
//Set sendmediain_to and/or sendmediaout_to JVoIP parameters to any port number. Let it be 50555.

public class AudioReceiver extends Thread {

  private DatagramSocket socket;
  private boolean running;
  private byte[] buf = new byte[1500];

  public AudioReceiver() {

  }

  public void run() {
    try{
```

```
        running = true;
        socket = new DatagramSocket(50555); //listen on the same port what you set for sendmediain_to and/or sendmediaout_to

        while(running)
        {
           DatagramPacket packet = new DatagramPacket(buf, buf.length);
           socket.receive(packet);

           InetAddress address = packet.getAddress();
           int port = packet.getPort();
           packet = new DatagramPacket(buf, buf.length, address, port);

           ProcessAudioPacket( packet.getData(),packet.getLength());

        }
        socket.close();
      }
      catch (Throwable e) {
        e.printStackTrace();
      }

   }

    public void terminate()
    {
      try{
          running = false;
          socket.close();
      }
      catch (Throwable e) {
        e.printStackTrace();
      }
    }

    public void ProcessAudioPacket(byte[] buff, int len)
    {
       //process the audio data here as you wish
    }
}

//create thread somewhere in your code: (new AudioReceiver()).start();
```

A simple working example can be downloaded from here.

5. PCM stream:

Use the default PCM stream if you don't have any ready to use software and you have to write it yourself.
Note: The packet emitted by the SIP media stack can be processed as-is by any audio player.

Example for stream playback using the Java built-in audio:

```
//import required libraries
import java.io.*;
import javax.sound.sampled.*;

//initialize audio
AudioFormat format = new AudioFormat(8000, 16, 1, true, false);  //you need to use 8000 for narrowband input or 16000 for wide-band
DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
SourceDataLine line = (SourceDataLine)AudioSystem.getLine(info);
line.open(format);
line.start();

//playback (you can pass the UDP packets as-is as received from JVoIP)
//this is usually called many times from your UDP socket read thread or on UDP packet received event
line.write(buffer, 0, length);

//close
line.flush();
line.stop();
line.close();
```

Note:
You must add proper exception handling for the above code (try/catch). Exceptions might occur if there is no suitable playback audio device.
More details about Java audio can be found here.
If you are interested in both streams (in/out) then you will need 2 separate lines (don't just feed both streams to the same line).
The same can be done in a very similar way in any programming language (just search for its audio playback module/interface/functions).

6. **Wave files:**

If you don't need real-time audio, then just use the API_VoiceRecord function and/or voicerecording and related parameters to obtain voice files in wav or other formats at the end of each call as described here.

7. **Barge-In:**

If you just wish to listen into conversation, then you can use the JVoIP itself for the job. Its barge-in feature allows you to bare into any call and create a hidden conference endpoint, so you can hear the conversation. This is often used in callcenters by supervisors to monitor the agents activity.
You can barge-in or spy on the calls by sending a specific SIP header specified by the bargeinheader parameter.
For example if you specify the value as "X-Barge", then when your server sends this in the INVITE (like for example "X-Barge: yes"), the call will be auto-accepted and hidden joining a conference with all calls made by the user/agent.
With other words: you just need to send a specific extra SIP header with the INVITE to barge-in. This header must match the JVoIP bargeinheader parameter and then JVoIP will silently accept the call without notifying the user about it and the caller party will be able to hear all the conversation made by the JVoIP user.

8. **SIP media streaming:**

Use the API_PlaySound function if you need to stream an audio file to a remote SIP endpoint.

Quality tests

JVoIP can be also used to test your VoIP server/carrier call quality by setting the aqtest parameter to 1.
For this a call have to be done between two extensions via your server (both extensions handled by JVoIP).
Launch the java sip client app like this:
JVoIP.jar serveraddress=SERVERIP username=USR1 password=PWD1 callto=USR2 sereveraddress2=SERVERADDRESS username2=USR2 password2=PWD2 aqtest=1 loglevel=5
Accept the incoming call and keep it for a while then hangup and check the logs by searching for "aqstat display begin". You will find a line like this:
EVENT,aqstats: avg delay: 51 msec, max delay: 56 msec, loss: 0%, cseqproblems: 0%, expected: 30069, recv: 30069, cseqerr: 0, err: 0
Fields:

- avg delay: total average RTP roundtrip time to server and back in milliseconds (should be only slightly more than ping time to your server)
- max delay: maximum rtp roundtrip time (should be not too bigger than avg delay)
- loss: lost packets percent (should be 0% if you have a good internet connection. Audio quality will drop considerably above 4%)
- cseqproblems: unordered packets and other issues in percent (should be 0%)
- expected: number of expected packets (depends on the test call duration)
- recv: number of received packets (should be the same as expected or slightly less)
- cseqerr: unordered packets and other issues
- err: unrecognized received packets

After this line you will see packet arrival timing statistics like this (this might be incorrect if the packets are too spread):
EVENT,recv in 50 msec: 3030 packets
EVENT,recv in 60 msec: 26680 packets
EVENT,recv in 70 msec: 246 packets

How to get logs?

This FAQ point explains How to generate and send logs from JVoIP.

Set the loglevel parameter to 5 and the logs will be written in the webphonelog.dat file (mwphonedata subfolder).

Details:

If you run into any issue with the VoIP SDK, Mizutech support most probably will ask you to send a problem description and detailed logs about the problem. Make sure that the JVoIP loglevel parameter is set to 5, reproduce the problem, and send the webphonelog.dat file (from the mwphonedata subfolder) attached to your email.

1) Make sure to **set the loglevel parameter to 5** and then reproduce the problem.

2) Once you reproduced the problem, you can grab the **logs** using any of the following methods:
   - if you haven't disabled the GUI then a log window should appear with loglevel 5. Copy its content with the Ctrl+A, Ctrl+C, Ctrl+V key combinations (related parameter: logview)
   - or copy the content of the console if you started the JVoIP from command line (related parameter: logtoconsole)
   - or copy the content of the java console content (related parameter: logtoconsole)
   - or send the generated log file: the webphonelog.dat file from the mwphonedata subfolder or any other *log.dat files (related parameter: canlogtofile)
     Note: if the folder with the JVoIP.jar is read-only then the mwphonedata subfolder might be created elsewhere. Use the API_GetWorkdir() function to get the path.
   - or you can collect the logs also via the LOG notifications if you set the events parameter to 3
   - or you can collect the logs by setting the logpolling parameter to 1 and use the API_GetLogs() function to get the logs as strings

3) **Send** the logs to webphone@mizu-voip.com as email file attachment with an accurate **description** of the issue (What you did? What happened? What should happen?).

   If the problem is call related, then make sure to have only one call in the log if possible (That one in which the problem was reproduced), otherwise let us know the SIP Call-ID of the problematic call.

   Make sure to send the whole log from the very beginning, not only the relevant part (including both the JVoIP startup and the disconnect of your test call if any).

   If possible or relevant, send also step-by-step instructions for the reproduction and one or two SIP accounts valid on your server which can be used by Mizutech support for testing.

Resources

VoIP SDK home-page: https://www.mizu-voip.com/Software/SIPSDK/JavaSIPSDK.aspx
Download (demo package): https://www.mizu-voip.com/Portals/0/Files/JVoIP.zip
Test/Example: https://www.mizu-voip.com/Portals/0/Files/JVoIPTest.java.txt
Media enhancements: https://www.mizu-voip.com/Portals/0/Files/mediaench.zip
Documentation:
- PDF: https://www.mizu-voip.com/Portals/0/Files/JVoIP.pdf
- HTML: https://www.mizu-voip.com/Portals/0/Files/documentation/jvoip/index.html
- WinHelp: https://www.mizu-voip.com/Portals/0/Files/JVoIP.chm

JavaDoc:
- Download: https://www.mizu-voip.com/Portals/0/Files/jvoip_javadoc.zip
- Online: https://www.mizu-voip.com/Portals/0/Files/jvoip_javadoc/index.html

For help, contact webphone@mizu-voip.com