LOXONE Smart Home

COMMUNICATING WITH THE LOXONE MINISERVER

Loxone Config

In order to modernize our interface, we improved our Miniserver, allowing it to communicate with WebSocket clients that conform to the RFC6455. This document explains what is to be done, in order to communicate with our Miniserver on the basis of an RFC6455 websocket. Parts of this document will cover communicating via HTTP-Requests too (e.g. Secured Commands, Hashing or the Structure File).

Arguments in commands are wrapped in curly brackets, e.g.: “jdev/test/{the argument}”. So when {the argument} is “yeah” it will look like this: “jdev/test/yeah” - no more curly brackets here.

Table of contents

Config 8.3

Setting up a connection

What do we need?

- a websocket client implementation conforming to the RFC6455
- the IP or URL of the Miniserver (including the port)
    - beware, if the Miniserver is using the Loxone CloudDNS service, you need to resolve it first. The websocket-redirect won't work. (see CloudDNS)
- valid credentials (user & password)

Step-by-step guide (see **Encryption** for even more secure connections)

1. ensure the Miniserver is reachable
    a. our apps are using an HTTP-Request that returns both the Miniservers Mac-Address and it's config version: "**{ipOrUrl}:{port}/jdev/cfg/api"**
    b. Alternatively "jdev/cfg/apiKey" may be used. It will too return a key for hashing.
2. open a WebSocket connection
    a. use the following path: "**ws://{ipOrUrl}:{port}/ws/rfc6455**"
    b. specify "**remotecontrol**" as Sec-WebSocket-Protocol.
    c. wss:// is not supported by the Miniserver.
3. request a key that will be used to hash your credentials

   a. send **“jdev/sys/getkey”** as message to the Miniserver
   b. Beware: the getkey-Response will be prequeled by a binary message containing our message-header (see MessageHeader), so the second message will contain the answer with the key.
4. send the hashed credentials to the Miniserver
   a. send “**authenticate/{hash}**”
   b. {hash} is the string “**{user}:{pass}**” hashed with the key returned by the getkey-Request using HMAC SHA1.
   c. for the detailed hashing process see Hashing
5. You should receive a response message with the code 200. Now your socket is up and running.

What can go wrong?

- If you provide invalid credentials you’ll receive a message with the response code 401.
- If you don’t authenticate within the first few seconds after establishing the socket connection, you’ll receive a message with the response code 420 and the socket will close.
- If you do not authenticate, but try to send any other command but the getkey- and authenticate-commands, you will receive a response with code 400.
- If you are blocked due too many failed login attempts, the Miniserver will block you for a certain time. The WebSocket closes right after it opens with a Close Code of 4003

General Info

In order to communicate with the Miniserver, a few things have to be known. This is a short textual explanation of the communication, details can be found further below.

CloudDNS

You can request the current IP Address of your Miniserver using an HTTP Request: *“dns.loxonecloud.com/?getip&snr={ SNR of Miniserver }&json=true”*. The Request returns the current IP Address and Port of the Miniserver represented as JSON.

Controls

In the following document a “Control” is a Function/Block-Function like “Virtual Input”, “Virtual State”, “Intelligent Room Controller”, “Sauna” etc.

Commands

The Miniserver will answer every command it receives, it will return a TextMessage as

confirmation. This message contains the command it has received, an HTTP-Status-Code as success indicator and a value. The value is the info that was requested or the state after a control-command was executed - but beware, these are not fully implemented, so instead rely on the state updates.

- a control command looks like
  - "*jdev/sps/io/{uuid}/{command}*"
- all commands can either start with "*jdev/*" or just "*dev/*"
  - *"jdev/"* in JSON-Format (Beware, the answer will contain 'dev' instead of 'jdev')
  - *"dev/"* delivers responses in XML-Format and is deprecated

Encryption

Available since 7.4.4.14

The hardware specifications of the Miniserver do not allow full featured SSL encryption, it would put the CPU under too much pressure leading to delayed responses to events. In order to still be able to send data to the Miniserver in a secure way (besides hashing), Command Encryption was introduced.

Secured Commands

We have the possibility to use a "visualization password" for Controls (set in Loxone Config), those passwords are added to the commands as described below:

1. request the visualization password from the user
2. request a Hashing-Key from the Miniserver ("*jdev/sys/getkey*")
3. hash the password (see Hashing)
4. send "*jdev/sps/**ios/{hash}**/{uuid}/{command}*"
   a. response has Code 200 if password was correct and command could be executed
   b. a response with Code 500 means the password was incorrect

Hashing

1. the key from the "*jdev/sys/getkey*" response is hex-encoded
2. create a hash from the desired text (user:pass or visualization password) using HMAC-SHA1
3. encode the hash back to hex

States

In order to receive the states of sensors and actuators, state-updates need to be enabled on the socket. This is done by sending "**jdev/sps/enablebinstatusupdate**" to the Miniserver. The Miniserver will only publish the states of sensors and actuators that are used in the user interface.

After this command is sent, the client will receive large initial Event-Tables containing Value-, Text-, Daytimer- and Weather-States. These Event-Tables contain the current states of all sensors and actuators.

The client needs to store these infos for the time the connection is active, because further on the Miniserver will only inform the client on changes to these states.

These states are always sent out as an UUID paired with some sort of value, using the Structure-File (see LoxAPP3.json) and the UUID you can find out what value belongs to which control.

Understanding the messages

As mentioned in the chapter on how to setup a connection, messages sent by the Miniserver are always prequeled by a binary message that contains a MessageHeader. So at first you'll receive the binary Message-Header and then the payload follows in a separate message.

Message Header

The message header is used to distinguish what kind of data is going to be sent next and how large the payload is going to be. In some cases, the Miniserver might not know yet how large the payload is going to be. For these cases a flag indicates that the size is estimated (see Estimated). The header is sent as a separate data packet before the actual payload is transmitted. This way the clients know ahead how large the payload is going to be. Based on this info, clients know how long it's going to take and respond accordingly (UI or timeouts).

Structure of the Message Header

The Message Header is an 8-byte binary message. It always starts with 0x03 as first byte, the second one is the identifier byte, which gives info on what kind of data is received next. The third byte is used for information flags and the fourth byte is reserved and not used right now. The last 4 bytes represent an unsigned integer that tells how large the payload is going to be.

1st Byte	2nd Byte	3rd Byte	4th Byte	5th Byte	6th Byte	7th Byte	8th Byte
0x03	Identifier	InfoFlags	rsvd	len	len	len	len

```
typedef struct {
    BYTE cBinType;          // fix 0x03
    BYTE cIdentifier;       // 8-Bit Unsigned Integer (little endian)
    BYTE cInfo;             // Info
    BYTE cReserved;         // reserved
```

Config 8.3

```
    UINT nLen;              // 32-Bit Unsigned Integer (little endian)
} PACKED WsBinHdr;
```

1st Byte: fix 0x03

2nd Byte: Identifier

8-bit Unsigned Integer (little endian)

The identifier byte is used to distinguish between the different kinds of messages

Identifer	Message-Type
0	Text-Message
1	Binary File
2	Event-Table of Value-States
3	Event-Table of Text-States
4	Event-Table of Daytimer-States
5	Out-Of-Service Indicator - presumably due to an Firmware-Update. No message is going to follow this header, the Miniserver closes the connection afterwards, the client may try to reconnect.
6	Keepalive response (after sending “keepalive”, the Miniserver will respond with this identifier - therefore the connection is up and running!)
7	Event-Table of Weather-States

3rd Byte: Info

The 3rd Byte of the Header is used to provide additional information regarding the incoming message.

1st Bit	2nd Bit	3rd Bit	4th Bit	5th Bit	6th Bit	7th Bit	8th Bit
Estimated	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd

1st Bit: Estimated

In order to get fast info of how big the next incoming data will be, a Header with the Estimated-Bit

---

set, tells, that the given size is only estimated (eg. Gateway Miniservers, ..)

An Estimated-Header is **always** followed by an exact Header to be able to read the data correctly!

3rd Byte: reserved

4th-8th Byte: length of payload
32-bit Unsigned Interger (little endian)
The size of the payload, may be estimated. This info can be used to adopt timeouts etc.

The actual messages

Text-Messages
Text-Messages are supported since day one of WebSockets. They are handled by all WebSocket implementations out there - so having a separate message-header telling there will be a Text-Message wouldn't be necessary. But in order to stay consistent in our application protocol, we did add it for those messages as well.

Text-messages are received as responses to commands, but our Structure-File and other XML- or JSON-Files are sent as Text-Messages too.

Binary Files
If you download files (e.g. images, statistic-data) from the Miniserver, you will receive a binary file. As mentioned before, files with text-content (e.g. the LoxAPP3.json) will be be delivered as a text-message, so you don't have to decode it.

Event-Tables
Incoming events are always grouped as tables according to their type (Value, Text, Daytimer, Weather). Each Event-Entry in these tables has it's own UUID, so you can assign the values to the correct Controls.

Event-Table of Value-States
Value-States are the simplest form of a state update, they consist of one UUID and one double value each, so their size is always 24 Bytes.

Binary-Structure of a Value-Event
```
typedef struct {
    PUUID uuid;      // 128-Bit uuid
    double dVal;     // 64-Bit Float (little endian) value
} PACKED EvData;
```

Event-Table of Text-States

Text-States are more complex since their size varies based on the text they contain. That is why they do not only consist of an UUID and the text, but also an unsigned int that specifies how long the text is.

- The UUID-Icon is used by the "Status"-Control (see Icons)
- If textLenght is not a multiple of 4 then padding bytes are appended, that are to be ignored.

Binary-Structure of a Text-Event

```
typedef struct {    // starts at multiple of 4
    PUUID uuid;                 // 128-Bit uuid
    PUUID uuidIcon;             // 128-Bit uuid of icon
    unsigned long textLength;   // 32-Bit Unsigned Integer (little endian)
    // text follows here
} PACKED EvDataText;
```

Event-Table of Daytimer-States

Like Text-States, Daytimer-States do not have a fixed size, it varies on how many Daytimer-Entries there are per Daytimer.

"nEntries" tells the number of daytimer-entries which the package contains.

Analog Daytimer: each entry does have it's value
Digital Daytimer: an existing entry means "on", no entry means "off"

Binary-Structure of a Daytimer-Event-Table

```
typedef struct {
    PUUID uuid;           // 128-Bit uuid
    double dDefValue;     // 64-Bit Float (little endian) default value
    int nrEntries;        // 32-Bit Integer (little endian)
    // entries (EvDataDaytimerEntry) follows here
} PACKED EvDataDaytimer;
```

Binary-Structure of a Daytimer-Entry

```
typedef struct {
    int nMode;  // 32-Bit Integer (little endian) number of mode
    int nFrom;  // 32-Bit Integer (little endian) from-time in minutes since midnight
    int nTo;    // 32-Bit Integer (little endian) to-time in minutes since midnight
    int bNeedActivate;     // 32-Bit Integer (little endian) need activate (trigger)
    double dValue;         // 64-Bit Float (little endian) value (if analog daytimer)
} PACKED EvDataDaytimerEntry;
```

Event-Table of Weather-States

If an active Weather-Abo is up and running, we also get Weather-State Updates.

Each Weather-Event-Table contains info about the up-to-dateness (in Seconds since 2009, UTC) of

the Weather-Information, the number of entries and the entries itself.

Binary-Structure of a Weather-Event-Table

```
typedef struct {
    PUUID uuid;                   // 128-Bit uuid
    unsigned int lastUpdate;      // 32-Bit Unsigned Integer (little endian)
    int nrEntries;                // 32-Bit Integer (little endian)
    // entries (EvDataWeatherEntry) follows here
} PACKED EvDataWeather;
```

Binary-Structure of a Weather-Entry

```
typedef struct {
    int timestamp;                // 32-Bit Integer (little endian)
    int weatherType;              // 32-Bit Integer (little endian)
    int windDirection;            // 32-Bit Integer (little endian)
    int solarRadiation;           // 32-Bit Integer (little endian)
    int relativeHumidity;         // 32-Bit Integer (little endian)
    double temperature;           // 64-Bit Float (little endian)
    double perceivedTemperature;  // 64-Bit Float (little endian)
    double dewPoint;              // 64-Bit Float (little endian)
    double precipitation;         // 64-Bit Float (little endian)
    double windSpeed;             // 64-Bit Float (little endian)
    double barometicPressure;     // 64-Bit Float (little endian)
} PACKED EvDataWeatherEntry;
```

The UUID

The Miniserver uses UUIDs in order to uniquely identify controls, in- or outputs. That is why the states that are published using Event-Tables have one UUID for each state, so that the values can be linked to their controls. Each UUID has a fixed size of 128 Bit.

Structure-Files such as “LoxAPP2.xml” (deprecated) or the new, tidied up “LoxAPP3.json” are providing the information on what UUID is related to what control, or to what in- or output. They can be acquired by sending “data/LoxAPP2.xml” or “data/LoxAPP3.json” to the Miniserver.

Binary-Structure of a UUID

```
typedef struct _UUID {
    unsigned long Data1;          // 32-Bit Unsigned Integer (little endian)
    unsigned short Data2;         // 16-Bit Unsigned Integer (little endian)
    unsigned short Data3;         // 16-Bit Unsigned Integer (little endian)
    unsigned char Data4[8];       // 8-Bit Uint8Array [8] (little endian)
} PACKED PUUID;
```

Converting a UUID to a string

```
CStringA str;
str.Format("%08x-%04x-%04x-%02x%02x%02x%02x%02x%02x%02x%02x",
    uuid.Data1,uuid.Data2,uuid.Data3,uuid.Data4[0],uuid.Data4[1],uuid.Data4[2],
    uuid.Data4[3],uuid.Data4[4],uuid.Data4[5],uuid.Data4[6],uuid.Data4[7])
```

Config 8.3

Icons

In general, icons are used for rooms and categories, but also for displaying states, as in the “Status”-Control. UUIDs are already used to link states to controls and furthermore they are also used to identify icons and link them to groups (rooms or categories), or “Status”-Controls.

SVG

Along with the release of Loxone Config 6.0 a new format for icons was introduced: “Scalable Vector Graphics”, short “svg”. Previously the only image format supported was “Portable Network Graphics”, short “png”. SVGs are image descriptions in XML-Format, they are not only scaleable losslessly, but they can also be modified and animated.

Where to get them, where to put them

The structure file gives info what icon is to be used where. In new configurations you will mostly find “svg”-Icons (e.g.: “00000000-0000-0020-2000000000000000.svg”). Since some Miniservers might make use of customized icons, there will still be some PNGs out there, even with newer Config-Versions. Those can be identified either by “.png” or the file appendix is simply missing (e.g. “00000000-0000-0020-2000000000000000”).

Images can be downloaded over the WebSocket by simply sending the UUID plus the type to the Miniserver. So “00000000-0000-0020-2000000000000000.svg” will return the SVG-File of this image, while “00000000-0000-0020-2000000000000000.png” will return the same image as PNG.

Text-States only contain UUIDs for icons without specifying the format. The only way to identify whether it is a PNG or an SVG is the response type given in the Message-Header when downloading the Icon. If it’s an Text-Message, then it’s an SVG (since it’s basically an XML-File), otherwise it’s an PNG and the Message-Header will indicate that it’s a binary file. So for all “Status”-Controls try to download an SVG and later decide based on the response whether it is PNG or SVG.

Caching

An icon is identified by an UUID per Miniserver. This UUID doesn’t change as long as the icon remains the same, regardless if you’re connected locally or remote. So you can reuse the downloaded icons locally and remote.

Structure-File: LoxAPP3.json

The Structure-File was mentioned a few times before in this document, it is the central element for

creating a visualisation. It contains almost everything you need to know about the Miniserver.

Download and caching

The new structure-file can be downloaded by sending “data/LoxAPP3.json” to the Miniserver. Since it’s a text-file, the Miniserver will respond with a Message-Header on the websocket accordingly.

The Structure-File has got a field “lastModified that contains a timestamp, this is the date when the configuration of the Miniserver was last changed. So every time a connection to a Miniserver is established, the first thing to do after the WebSocket is up and running is to check if your cached version of the Structure-File is up to date. This is achieved by sending “jdev/sps/LoxAPPversion3” to the Miniserver and comparing its response to the value of the “lastModified” field of your cached Structure-File.

More info

For detailed information on the Structure File & the controls within please see the separate document on the Structure File.

TaskRecorder

The TaskRecorder can be used to schedule a set of commands to be executed at a certain time. The commands will be executed in the same timely order as they were recorded (not all commands at once) starting at the desired point of time.

Tasks are not published using state updates, they need to be kept up to date on the ui via polling. Tasks consist of one or more single commands. A set of commands with the same name are building a task. Each command has its own date and time.

To delete a whole task, every single command in the task needs to be removed separately. To update a task, all commands are removed and added again

Formats

- <date> 2014-07-22 19:02:00
- <name> String
- <command> eg. “0a3d639e-00f1-141c-ffffeee0009800b5/1/off”

Commands

- jdev/sps/listcmds
    - returns all commands, separated by “, ”

- jdev/sps/removecmd/{date}/{name}/{command}
- jdev/sps/addcmd/{date}/{name}/{command}
- jdev/sps/addscmd/{date}/{hash}/{name}/{command}
    - adds a command with visu password
    - hash must be generated with a key (from get key) and the visu password

Command Encryption

Available since 8.1

As mentioned in the introduction section on encryption, Command Encryption is a technique that allows clients to encrypt commands that would usually be sent via plain text. It is based on AES256 and Public-key cryptography for the AES session key exchange.

A few commands aren't supported:

- Images/Icons (.svg, .png, camimage)
- Files (LoxAPP3.json, fsget/fslist)
- Statistic Files and Data

RSA	**AES**
ECB, PKCS1, Base64 with NoWrap	CBC, ZeroBytePadding, Base64 with NoWrap, 16 Byte IV

Step-by-step Guide HTTP Requests

1. Acquire the Miniservers public key via "jdev/sys/getPublicKey" -> {publicKey}
    a. Store on the client
    b. Format: X.509 encoded key in ANS.1
2. Prepare your command -> {cmd}
    a. If authentication is required, append "?auth={hash}" to the {cmd} (see Hashing)
    b. Also append the user as a request parameter "&user={username}"

3. Generate a random salt, hex string (length may vary, e.g. 2 bytes) -> {salt}
4. Prepend the salt to the actual message “salt/{salt}/{cmd}” ->{plaintext}
5. Generate a AES256 key -> {key} (Hex)
6. Generate a random AES iv (16 byte) -> {iv} (Hex)
7. Encrypt the {plaintext} with AES {key} + {iv} -> {cipher} (Base64)
8. URI-Component-Encode the {cipher} -> {enc-cipher}
9. Prepare the command-> {encrypted-command}
    a. “jdev/sys/enc/{enc-cipher}”
        i. only the command itself is encrypted
    b. “jdev/sys/fenc/{enc-cipher}”
        i. The Miniserver also AES Encrypts the response (Base64)
        ii. The mime-type is the one from the decrypted response
10. RSA Encrypt the AES key+iv with the {publicKey} -> {session-key} (Base64)
    a. “{key}:{iv}”
11. URI-Component-Encode the {session-key} -> {enc-session-key}
12. Append the session key to the {encrypted-command} -> {encrypted-command}
    a. “{encrypted-command}?sk={enc-session-key}”
13. Send the request!
    a. This request doesn’t require credentials itself, the {cmd} may contain the credentials if needed
14. The Miniserver will decrypt and process the command.
    a. If it cannot be decrypted (invalid public key, unexpected salt change) it will return 401
15. The Miniserver will respond after the decrypted command was processed and return the value & status code as with a regular command.

Step-by-step Guide WebSocket Connections:

You need to authenticate slightly different in order to use encrypted commands over WebSocket

1. Acquire the Miniservers public key via “jdev/sys/getPublicKey” -> {publicKey}
    a. Store on the client
    b. Format: X.509 encoded key in ANS.1
2. Open WebSocket
3. Generate a AES256 key -> {key} (Hex)
4. Generate a random AES iv (16 byte) -> {iv} (Hex)
5. RSA Encrypt the AES key+iv with the {publicKey} -> {session-key} (Base64)
    a. “{key}:{iv}”
6. Exchange keys via “jdev/sys/keyexchange/{session-key}”
    a. Returns a key like “jdev/sys/getkey” encrypted with AES

7. AES Decrypt the received key with the session key
8. Create the authentication hash (see Setting up a connection) -> {hash} (Hex)
9. AES Encrypt the {hash}/{username} -> {cipher}
10. Authenticate yourself via "authenticateEnc/{cipher}"
11. WebSocket ready..
12. Generate a random salt, hex string (length may vary, e.g. 2 bytes) -> {salt}
13. AES Encrypt the command "salt/{salt}/{cmd}" -> {cipher}
14. Prepare the command-> {encrypted-command}
    a. "jdev/sys/enc/{enc-cipher}"
        i. only the command itself is encrypted
    b. "jdev/sys/fenc/{enc-cipher}"
        i. The Miniserver also AES Encrypts the response value (Base64)
        ii. The mime-type is the one from the decrypted response
15. Send the command!
16. Update the salt on your behalf, eg. every hour. This prevents replay attacks on compromised websocket connections.
    a. {cipher}: "nextSalt/{prevSalt}/{nextSalt}/{cmd}" encrypted with the session key (AES)

Handling an active connection

Keeping the connection alive

The Miniserver has to watch over it's clients and has to keep them all informed on everything that's changed. In order to prevent sending updates to clients that aren't listening anymore, it will close the connection if the client doesn't send anything for more than 5 minutes.

To prevent this, while not having to constantly query a control or alike, there is a special command called "keepalive". Whenever a client sends this command to the Miniserver, it will respond with a Message Header with the identifier 0x06. This command can be used to tell the Miniserver that the client is still there and listens on the WebSocket.

Detecting issues

Our websocket is used for remote control. Mostly this is being done by apps running on smartphones that don't always enjoy the best connection quality (poor carrier network, WiFi almost out of reach). A poor or broken connection might cause that the user looks at old outdated data. E.g.: the app could show that your garage door is closed, while it's fully open.

By repeatedly sending out the keepalive-Request, the time between request and response (8 byte Message-Header) can be used as an indicator for the connection quality. When the Miniserver

sends large messages this might lead to mistakenly detecting a connection problem, since receiving the response might take a while. The payload size in the Message-Headers can be used to adopt timeouts accordingly.