

# CTRL – IoT

## Contents

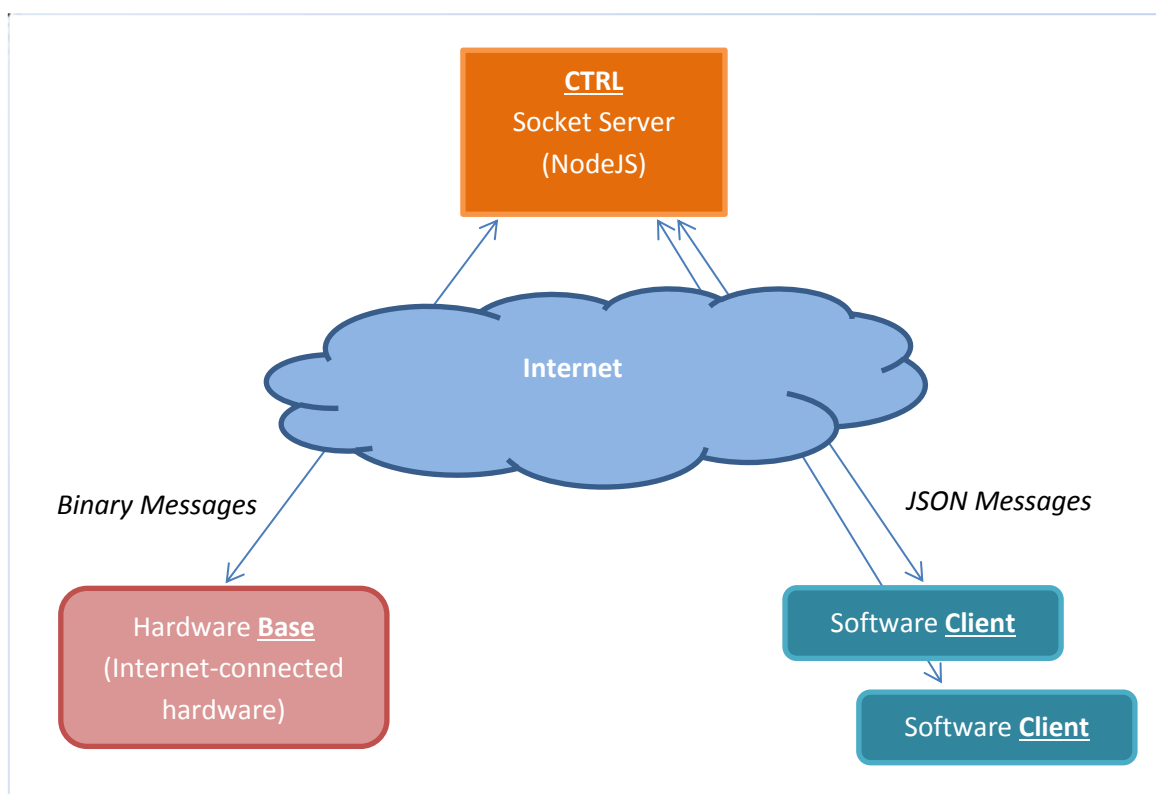
CTRL – IoT .....	1
Introduction.....	2
Message Exchange Protocol.....	3
JSON Messages .....	4
Header – section.....	4
TXsender – section .....	5
Data – section .....	5
Binary Messages .....	6
Length – section .....	6
Header – section.....	6
TXsender – section .....	6
Data – section .....	6
Authenticating the Socket Connection.....	7
Base Authentication Procedure.....	7
Client Authentication Procedure.....	7
Synchronization Mechanism (Sequence Synchronization) .....	9
Special System Messages .....	11
Special System Messages sent to Client.....	11
Base Connection Status .....	11
Special System Messages sent to Base.....	11
Document Change Log .....	12

## Introduction

The idea is to connect the Base to the Server and forward incoming message to all connected Clients associated to that Base and vice versa. There can be multiple Clients associated to one Base, but any one Client can only be associated to just one Base.

System consists of these parts:

1. Base Station (**Base**) – Internet-connected hardware
2. Software Client (**Client**) – Internet-connected software: Web app, Android app...
3. Socket Server (**Server**) – Server written in NodeJS that accepts connections from many Bases and Clients (some might know it as *the Cloud*)



*Basic idea (shows just one Base with multiple Clients for **that** Base)*

## Message Exchange Protocol

Important aspect of message forwarding is to ensure that messages which are forwarded through Server **are delivered** to their destination **in order** they were sent out from the sender. Even though TCP sockets are used for communication, there is no “out of the box” mechanism to ensure that sending party knows that message went through and was delivered to Server. TCP will generate a *timeout* in case connection breaks but until that happens sender doesn't know what has been delivered to the Server and what hasn't.

Message Exchange Protocol features:

- Ensures message delivery by using message queues
- Handles acknowledgements of received messages
- Makes sure that message re-transmissions are ignored but acknowledged back
- Supports notification-type messages which are not acknowledged back and not re-transmitted in case of failure of delivery
- Supports system-type messages which are not forwarded to other party (from Base -> Client and vice versa) which are used for system-related operations (for example: Base can privately communicate with Server and ask for current Timestamp for its internal RTC)
- Supports “Back-off” acknowledgements with exponential delay increments to inform the sender to delay sending further messages (implemented only for Binary Messages for communication between Server<->Base)

Server talks to Clients by using JSON Messages and to Bases by using Binary Messages. Talking to Base uses Binary Messages because Bases are usually small micro-controller solutions which do not have plenty resources to parse JSON, so Server is happy to bridge these two types of messages together.

Each type of message contains a Header, TXsender and data sections. **Header** section contains important bits about the message itself (whether it is an acknowledgement, a system-type message, or even a notification-type message and so on). **TXsender** is used for synchronization between sender and a receiver and is used to check whether received message is a re-transmission, new message, or if sender and receiver are out of sync. Payload in “**data**” section is binary data in case of Binary Messages and for JSON Messages it is in hexadecimal ASCII format!

Format of JSON and Binary Message is naturally different but meaning of sections and fields is the same.

## JSON Messages

This message type is a string of JSON with three sections: **header**, **TXsender** and **data**. Each message is terminated by a New-Line character (\n) so it is important to never introduce this character in message itself except at the very end where it actually ends. Section "data" is encoded in hexadecimal ASCII format and should always contain even number of hexadecimal characters.

Example of JSON Message:

```
{
  „header“: {
    „sync“: false,
    „ack“: false,
    „processed“: false,
    „out_of_sync“: false,
    „notification“: false,
    „system_message“: false,
    „backoff“: false
  },
  „TXsender“: 5081,
  „data“: „68656c6c6f20776f726c6421“
}
```

### Header – section

Property	Value	Description
<b>sync</b>	true/false	Tells the receiver of this message to sync to „0“. <u>Used only in authentication procedure when socket connection is first (re)created.</u>
<b>ack</b>	true/false	Means that this message is an acknowledgment of previous message with the same TXsender value provided.
<b>processed</b>	true/false	Tells whether the receiving side processed this command. (It is not processed only if it was a re-transmission). <u>This bit is used only if this message is an ACK.</u>
<b>out_of_sync</b>	true/false	Tells that receiver is out of sync with the transmitter of the message with this TXsender. <i>At this point this is not handled in neither Server nor Client. This is actually a non-recoverably sync situation that probably requires flushing entire „pending TX queue“, and starting from scratch.</i> <u>This bit is used only if this message is an ACK.</u>
<b>notification</b>	true/false	Low priority messages that

		don't get ACKs back, and no re-transmissions in case of failure. Also TXsender field is not checked for sync (it is ignored and can be omitted from the message).
<b>system_message</b>	true/false	Tells to receiving side that this message is a private message between connected party and the server (not forwarded to/from either Base or Client).
<b>backoff</b>	true/false	Tells the receiver that this TXsender message is not received, and to delay sending further messages. <i>Currently not implemented in Client&lt;-&gt;Server communication because we assume that both Server and Client have enough storage space and processing power.</i> <u>This bit is used only if this message is an ACK.</u>

#### ***TXsender – section***

This is the sequence ID of the transmitter. Its value increments from 1 to  $2^{32}$  (unsigned integer) and can only be reset to 1 during authentication procedure. This means that each connection can transfer  $2^{32}$  messages until it rolls over to 1.

#### ***Data – section***

This is the actual payload and is encoded in hexadecimal ASCII format. There is no actual limitation to the length of this data but Base can accept only first 65535 bytes (a bit less than that, which will be described in Binary Messages topic).

## Binary Messages

This message type is a binary data stream which consists out of four objects: **Length**, **Header**, **TXsender** and **data**. There is no “termination character” after each message (binary data stream) but there is a **length** section which tells how many bytes follow for that particular message.

Example of Binary Message:

00110000001B668656c6c6f20776f726c6421

Sections highlighted in example are:

- 0011 = **Length** section
- 00 = **Header** section
- 000001B6 = **TXsender** section
- 68656c6c6f20776f726c6421 = Section **data**

### Length – section

This section is always two bytes long and defines length of the message which follows. These two bytes are **not included** into the actual message length (in the example above we can see that entire binary data stream is actually 19 bytes long, but the **Length** section says 17 bytes).

*Note: Better idea would be to implement dynamic length instead of fixed 2-bytes for length. See MQTT protocol definition for this solution. This could be done in future versions.*

### Header – section

This section is 1 byte long and contains flags which are important for the transmitted message.

Flag (property)	Bit location	Description
sync	00000001	Same as in JSON Messages
ack	00000010	Same as in JSON Messages
processed	00000100	Same as in JSON Messages
out_of_sync	00001000	Same as in JSON Messages
notification	00010000	Same as in JSON Messages
system_message	00100000	Same as in JSON Messages
backoff	01000000	Same as in JSON Messages
reserved	10000000	

### TXsender – section

This is the sequence ID of the transmitter. Its value increments from 1 to  $2^{32}$  (unsigned integer) and can only be reset to 1 during authentication procedure. This means that each connection can transfer  $2^{32}$  messages until it rolls over to 1.

### Data – section

This is the actual payload data and is a raw binary stream. In current implementation the limitation to the length of this data is  $65535 - \text{sizeof}(\text{Header}) - \text{sizeof}(\text{TXsender})$ .

## Authenticating the Socket Connection

After the socket connection is established from either Base or Client towards the Server, it must be authenticated. In case it doesn't get authenticated within a timeout, Server will close the connection. Bases authenticate by using their **baseid** and Clients authenticate by **username** and **password**. Each unsuccessful authentication attempt is logged on Server and can be viewed. There is also a protection against brute-force password attacks for both Base and Client accounts. Server checks for failed authentication attempts within past 5 minutes (configurable) for IP address which is trying to authenticate and in case of 5 failed authentication attempts (also configurable) it refuses the authentication attempt completely.

## Base Authentication Procedure

After Base establishes the socket connection to Server, it must send the authentication request message. Example of authentication request message (in hexadecimal format):

0015HH00000000BABABABABABABABABABABABABABABABABA

Explanation of sections:

- 0015 = length of message, always 0x15 (21 decimal)
- HH = **Header** section (please see *Synchronization Mechanism* for importance of Header section bits during authentication procedure)
- 00000000 = **TXsender** section (not important during authentication procedure)
- BABABABABABABABABABABABABABABABA = **data** section, 16 bytes of your Base's **baseid**

After Server receives and processes the received authentication request message, it will reply with an authentication reply message. Example of authentication reply message (in hexadecimal format):

0006HH00000000RR

Explanation of sections:

- 0015 = length of message, always 0x15 (21 decimal)
- HH = **Header** section, **notification** bit is always set, and **system\_message** bit is always set. (please see *Synchronization Mechanism* for importance of other Header section bits during authentication procedure)
- 00000000 = **TXsender** section (not important during authentication procedure)
- RR = **data** section, 1 byte of authentication result. Possible values that will be sent by the server are: 0x00 (AUTH **OK**) and 0x01 (AUTH **ERROR**).

## Client Authentication Procedure

After Client establishes the socket connection to Server, it must send the authentication request message. Example of authentication request message:

```
{
  „header”: {
    „sync”: (please see Synchronization Mechanism topic),
    „ack”: ignored,
    „processed”: ignored,
    „out_of_sync”: ignored,
    „notification”: ignored,
```

```

        „system_message”: ignored,
        „backoff”: ignored
    },
    „TXsender”: ignored,
    „data”: {
        „username”: „user1”,
        „password”: „secretpassword123”
    }
}

```

After Server receives and processes the received authentication request message, it will reply with an authentication reply message. Example of authentication reply message:

```

{
    „header”: {
        „sync”: (please see Synchronization Mechanism topic),
        „ack”: false,
        „processed”: false,
        „out_of_sync”: false,
        „notification”: true,
        „system_message”: true,
        „backoff”: false
    },
    „TXsender”: 0,
    „data”: {
        „type”: „authentication_response”,
        „result”: 0,
        „description”: „Logged in.”
    }
}

```

Section **data** contains object with three properties: **type**, **result** and **description**. Property **result** can have three values:

- 0 = Logged in.
- 1 = Wrong username/password.
- 2 = Too many failed authentication requests.

Property **type** is a string containing word “**authentication\_response**”.

Upon successful authentication of the Client, Server will send a Special System Message to inform the Client about the connection status of his associated Base. Please see *Special System Messages* topic for this.



## Synchronization Mechanism (Sequence Synchronization)

In order to implement re-transmissions and acknowledgements and to distinguish re-transmissions from new messages, a sequence number is introduced into the protocol. This sequence number is called TXsender, starts from 1 and increments by the sending party after each transmitted message. Maximum value it can get to is  $2^{32}$  (unsigned integer). In practice this sequence number will hardly ever reach its maximum value (and potentially rollover) because socket connection will probably break before it happens. After the socket connection breaks and re-connects, authentication procedure will take place where this sequence number gets a chance to reset (re-sync) to 1. In case it does get to its maximum value, socket connection must be restarted and re-synchronization will take place during authentication handshake. This sequence number has a capacity to send one message per second for 136 years until it rolls over, so this limitation is not to be worried about.

Each transmitting party has its own TXsender sequence number and is responsible for incrementing it and restarting to 1. Sequence numbers are not blindly restarted on each authentication handshake, but are restarted only if there are no pending messages on the sending party.

Socket connections are always originated by Base and Client towards the Server, so the handshaking procedure is as follows (example: Base is connecting to Server and authenticating...):

```
BASE OPENED A SOCKET CONNECTION TO SERVER:
1. Base prepares an authentication message to send to Server
2. Base checks to see if it has some pending messages to send to Server
3. If it has pending messages in queue:
    a. Base doesn't set SYNC bit in Header and doesn't reset its internal
       TXsender value to 1 but continues from whatever value it had earlier
4. If it doesn't have pending messages in queue:
    a. Base sets SYNC bit in Header, and resets its internal TXsender to 1
5. Base now sends the authentication message to Server and waits for the reply
6. ...
7. Server prepares an authentication reply message to send to Base
8. Server accepts the message and if authenticated it checks to see whether it
   must re-sync its copy of Base's sequence number to 0
    a. If SYNC bit is set in authentication request message:
        i. Server resets its copy of Base's TXsender to 0
    b. If SYNC bit is not set in authentication request message:
        i. Server doesn't alter its copy of Base's TXsender
9. Server now checks to see if it has some pending messages to send to Base
10. If it has pending messages in queue:
    a. Server doesn't set SYNC bit in Header of the reply message and doesn't
       reset its internal TXsender value to 1 but continues from whatever
       value it had earlier
11. If it doesn't have pending messages in queue:
    a. Base sets SYNC bit in Header, and resets its internal TXsender to 1
12. Server now sends the authentication reply message to Server and in case it
    had pending messages queue it will also start sending those pending messages
    to Base
13. Base naturally receives this authentication reply message and if it also had
    some pending messages in queue it will start sending them to Server
14. ...
```

*Pseudo-code*

If during communication either party receives an acknowledgement with out-of-sync bit set, that party should flush its pending messages queue and restart the socket connection that will re-synchronize sequence numbers.

For example: in case the Server receives out-of-sync acknowledgement, it should flush the queue and simply close the socket connection. The party to which Server was communicating will re-establish

the connection and continue with the authentication handshake which will re-synchronize the sequence number.

In case Base or Client receives the out-of-sync acknowledgement, it should flush the queue and restart the socket connection followed by the authentication and re-sync procedure as mentioned before.

## Special System Messages

This section describes special system messages which could be sent by Server (to Base and/or Client) at any given time on active socket connection.

## Special System Messages sent to Client

### Base Connection Status

A connected Client can receive a system message from the Server when Base connects or disconnects from its socket. This message is also of type **notification** and is not re-transmitted and doesn't have to be acknowledged by the Client.

Example of this system message notification:

```
{  
    „header“: {  
        „sync“: false,  
        „ack“: false,  
        „processed“: false,  
        „out_of_sync“: false,  
        „notification“: true,  
        „system_message“: true,  
        „backoff“: false  
    },  
    „TXsender“: 0,  
    „data“: {  
        „type“: “base_connection_status”,  
        „connected“: true,  
        „baseid“: „bababababababababababababababababababababa”  
    }  
}
```

Section **data** contains three properties: **type**, **connected** and **baseid**.

Property **type** will have word: "base\_connection\_status", property **connected** can have a Boolean value of true or false.

Since this is a message of type **notification**, the **TXsender** is ignored and will always have value: 0.

## Special System Messages sent to Base

There are currently no implemented special system messages which are sent to Base by the Server.

## Document Change Log

Date	Changed By	Summary
2014-09-08	Trax, <a href="mailto:trax@elektronika.ba">trax@elektronika.ba</a>	Initial document write
2014-09-09	Trax, <a href="mailto:trax@elektronika.ba">trax@elektronika.ba</a>	Still writing initial document...