# CTRL – IoT

## Contents


CTRL – IoT ..... 1
Introduction ..... 3
Message Exchange Protocol ..... 4
Security ..... 5
Client Security ..... 5
Base Security ..... 5
JSON Messages ..... 6
Header – section ..... 6
BaseID – section ..... 7
TXsender – section ..... 7
Data – section ..... 7
Binary Messages ..... 8
Length – section ..... 8
Header – section ..... 8
TXsender – section ..... 9
Data – section ..... 9
Special feature during acknowledgement of data received from Server ..... 9
Authenticating the Socket Connection ..... 10
Base Authentication Procedure ..... 10
Server-Stored TXserver ..... 11
Client Authentication Procedure ..... 12
Synchronization Mechanism (Sequence Synchronization) ..... 14
Special System Messages ..... 16
Special System Messages sent from Server -> Client ..... 16
Base Connection Status ..... 16
Special System Messages sent from Client -> Server ..... 16
Pull unacknowledged messages ..... 16
Special System Messages sent from Server -> Base ..... 17
Special System Messages sent from Base -> Server ..... 17
Pull unacknowledged messages ..... 17

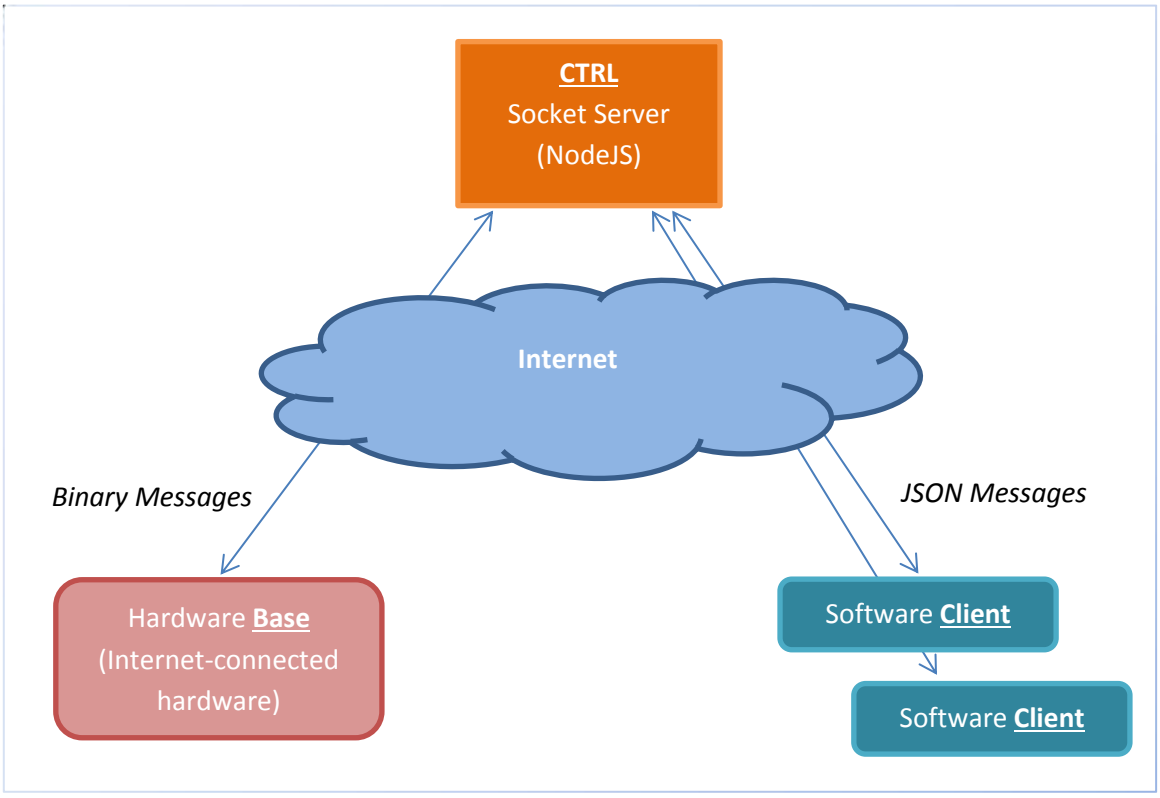
Enable Keep-Alive on current socket connection ..... 17
Disable Keep-Alive on current socket connection..... 17
Save variable to Server ..... 18
Read variable from Server ..... 18
Get Timestamp from Server ..... 18
Document Change Log ..... 19

## Introduction

The idea is to connect the Base(s) and Clients to Server and exchange messages between Base and all associated Clients currently connected to Server. All associated but unconnected Clients will receive messages from Base(s) once they connect to Server. There can be multiple Clients associated to one Base, and multiple Bases can be controlled by the same client. This is different from the previous version in which number of Clients could control 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)*

Client can send a message that will be delivered **to all** or **just one targeted** Base (which it owns) but Bases send messages to **all** associated Clients, meaning that Base can't target a particular Client for message delivery.

## 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 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 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 are the same.

## Security

All communication between Client<->Server and Base<->Server is encrypted.

### Client Security

Connection between Client and Server is secured using secure sockets with TLS. No custom work has been done here.

### Base Security

Connection between Base and Server is secured using AES-128 encryption in CBC mode. Ciphertext is protected by 16 bytes of AES-CMAC and appended to message which is being transferred. Structure of encrypted packet is:

**3000B48207A070C6380D1070C7C678292424BINARY_MESSAGE827A23C64E08734896712609864635278657930428**

Sections highlighted in example are:

- **3000** = **All-Length** of data that follows (in Little Endian byte-order)
- **B48207A070C6380D1070C7C678292424** = **IV** – random Initialization Vector different for each transmission, 16 bytes
- **BINARY_MESSAGE** = the actual **Binary Message** (see section **Binary Messages** for description)
- **827A23C64E** = padding of **BINARY_MESSAGE** to multiple of 16 bytes so that entire encrypted data is dividable by 16. This is any random data that will be discarded once packet gets decrypted at the receiving side
- **08734896712609864635278657930428** = 16 byte AES-CMAC protection of entire ciphertext

All fields (sections) are encrypted except the "**All-Length**" and AES-CMAC. AES-CMAC is used to ensure that receiving side doesn't decrypt the message in case it was tampered with (intercepted and changed). In case of tampering it will simply be discarded.

## JSON Messages

This message type is a string of JSON with three sections: **header**, **TXsender** and **data**. Optional fourth section is **baseid** which is used when sending message to just one targeted Base. 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
    },
    „baseid": ["01234567890123456789012345678901"],
    „TXsender": 5081,
    „data": „68656c6c6f20776f726c6421"
}
```

### *Header – section*

| Property | Value | Description |
|---|---|---|
| **sync** | true/false | Tells the receiver of this message to sync to „0". Used only in authentication procedure when socket connection is first (re)created. |
| **ack** | true/false | Means that this message is an acknowledgment of previous message with the same TXsender value provided. |
| **processed** | true/false | Tells whether the receiving side processed this command. (It is not processed only if it was a re-transmission). This bit is used only if this message is an ACK. |
| **out_of_sync** | true/false | Tells that receiver is out of sync with the transmitter of the message with this TXsender. The sender of message should try re-sending all unacknowledged messages from its queue, and if failed to re-sync after *n* attempts should flush entire TX queue. This bit is used only if this message is an ACK. |

| | | |
|---|---|---|
| **notification** | 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). |
| **system_message** | 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). |
| **backoff** | true/false | Tells the receiver that this TXsender message is not received, and to delay sending further messages.<br>*Currently not implemented in Client<->Server communication because we assume that both Server and Client have enough storage space and processing power.*<br>This bit is used only if this message is an ACK. |

*Note:* In case any property is missing from Header section it is considered to be **false**.

### *BaseID – section*

This is an array of BaseID tokens of Bases that will receive this message. Targeted Bases must be owned (associated to) by the Client in order to forward the message successfully.
In case "baseid" section is omitted from JSON Message or an empty array is provided, the message will be forwarded to every Base associated by the sending Client!
*Note:* "baseid" parameter can (and should) be omitted in Special System Messages for communication between Client and Server.

### *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, and later on might be extended to more by using MQTT length-coding method).

## 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. Please note that this is explanation of Binary Message once it is extracted from an encrypted communication!

Example of Binary Message:

`1100000101B6000068656c6c6f20776f726c6421`

Sections highlighted in example are:

- `1100` = **Length** section
- `00` = **Header** section
- `01B60000` = **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:* byte-order of this field is in Little Endian.

*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 version. http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html#fixed-header*

### *Header – section*

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

| Flag (property) | Bit location | Description |
|---|---|---|
| **Sync** | 0000000**1** | *Same as in JSON Messages* |
| **Ack** | 000000**1**0 | *Same as in JSON Messages* |
| **Processed** | 00000**1**00 | *Same as in JSON Messages* |
| **out_of_sync** | 0000**1**000 | *Same as in JSON Messages* |
| **notification** | 000**1**0000 | *Same as in JSON Messages* |
| **system_message** | 00**1**00000 | *Same as in JSON Messages* |
| **Backoff** | 0**1**000000 | *Same as in JSON Messages* |
| **save_TXserver** | **1**0000000 | Tells server to save provided value into DB. Value is provided as data with an acknowledgment and must be 4 bytes long. It gets sent back to Base once socket gets authorized. This bit is used only if this message is an ACK. |

### *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. *Note:* byte-order of this field is in Little Endian.

### *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-sizeof(Header)-sizeof(TXsender).

### *Special feature during acknowledgement of data received from Server*

Whenever Base receives a message from Server that needs to be acknowledged back, it can acknowledge with bit **save_TXserver** set and provide a 4 byte value as **Data** in that acknowledgement message. This data will be saved to Server's database and will be sent back to Base once it gets (re)authorized. This is a special feature of Server which allows Base to store its copy if TXserver on Server's premises. This is quite useful because this way Base doesn't have to write this value to its internal EEPROM or Flash memory after each acknowledgement.

## 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 **auth_token**. Each unsuccessful authentication attempt is logged on Server and can be viewed later. There is also a protection against brute-force 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 authorize. This is a challenge-response authentication and is done in two phases:

**Phase 1:** Base sends a Binary Message containing its Base ID (16 bytes) in **Data** section. Header and TXsender fields are not important in this Phase. This Binary Message is encrypted using zero-key (AES-128 Key of all 0x00 bytes). This doesn't pose a security risk though.

Example of authentication request message (in hexadecimal format):

```
1500 00 00000000 BABABABABABABABABABABABABABABABA
```

Explanation of sections:

- 1500 = length of message, always 0x15 (21 decimal) (in Little Endian byte-order)
- 00 = **Header** section (not important during Phase 1 authentication procedure)
- 00000000 = **TXsender** section (not important during authentication procedure)
- BABABABABABABABABABABABABABABABA = **data** section, 16 bytes of your Base's **baseid**

After the Server receives this Phase 1 message and if provided Base ID exists in the system it will reply with a challenge request. This challenge request will be encrypted using the encryption key of the provided Base read out from the database! Example of challenge message being sent from Server to Base (in hexadecimal format):

```
1500 00 00000000 FAFAFAFAFAFAFAFAFAFAFAFAFAFAFAFA
```

Explanation of sections:

- 1500 = length of Binary Message, always 0x15 (16 decimal) (in Little Endian byte-order)
- 00 = **Header** section (not important during Phase 1 authentication procedure)
- 00000000 = **TXsender** section (not important during authentication procedure)
- **FAFAFAFAFAFAFAFAFAFAFAFAFAFAFAFA** = **data** section, 16 bytes of challenge value. This is just a random value that will be returned back from Base but prepended with different 16 random bytes. Since the reply will be encrypted in AES-CBC mode and protected with AES-CMAC, it is safe to use this simple challenge-response mechanism

**Phase 2:** Base replies to this challenge with 32 bytes of data. First 16 bytes are any random bytes and last 16 bytes are the same challenge Base previously received. In this Phase the Header section is

important and it carries SYNC bit that will be read by Server. This message is encrypted using Base's secret key.

Example of response to challenge from Base to Server (in hexadecimal format):

**2500 HH 00000000 EAEAEAEAEAEAEA...EAEAEAEAEAEAEA**

Explanation of sections:

- 2500 = length of Binary Message, always 0x25 (37 decimal) (in Little Endian byte-order)
- HH = **Header** section (important field is SYNC, please see Synchronization Mechanism section)
- 00000000 = **TXsender** section (not important during authentication procedure)
- **EAEAEAEAEAEAEA...EAEAEAEAEAEAEA** = **data** section, 32 bytes of challenge response value. This contains 16 bytes of random value and 16 bytes of the challenge value

Now Server receives this response, and in case response is decrypted and validated the Server will send the last message in this authentication procedure. This message will contain SYNC bit in **Header** section which will be extracted by Base and it will also contain the Server-Stored TXserver value in **Data** section.

Example of last message from Server to Base (in hexadecimal format):

**0900 HH 00000000 ABCDEFAB**

Explanation of sections:

- 0900 = length of Binary Message, always 0x09 (9 decimal) (in Little Endian byte-order)
- HH = **Header** section (important field is SYNC, please see Synchronization Mechanism section)
- 00000000 = **TXsender** section (not important during authentication procedure)
- **ABCDEFAB** = **data** section, 4 bytes of Server-Stored TXserver value (in Little Endian byte-order)

Base will receive this message and treat the SYNC bit according to the Synchronization Mechanism. The special data in this message is the server-stored TXserver value. Please see Server-Stored TXserver section for explanation bellow.

In case an error happens during authentication procedure (wrong Base ID is provided or AES keys mismatch, the Server will close the connection). Base will know if it failed to authenticate only if connection breaks during the authentication procedure – there is no "failed to authenticate" message sent back from Server to Base (like there is when Client tries to authenticate).

### *Server-Stored TXserver*

CTRL protocol requires both communicating sides to keep track of synchronization values (TXsender) even after power loss. This poses a practical problem for Bases as they are usually hardware devices with Flash and/or EEPROM memory that can be worn out after many re-writes. To solve this problem, the Server can store this TXserver value in its DB. This value will be pushed to Base once it

gets authorized so it can continue counting it. The TXserver value is pushed to Server on each acknowledgement Base sends to it.

### 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": {
            „auth_token": „my-secret-auth-token"
    }
}
```

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 auth_token.
- 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:

**Phase 1**

1. Base sends its baseid to Server, entire communication is encrypted using zero-AES-key (all zeroes)
2. *...data transfer...*
3. Server receives baseid, validates and decrypts using zero-AES-key. If provided baseid value doesn't exist in database, Server closes the connection. If provided baseid does exist, then the Server loads this Base's secret AES-key from the database, generates random 16 bytes of challenge, encrypts and sends to Base

**Phase 2**

4. Base receives this challenge, verifies and decrypts using its secret AES-key stored in its memory. It now generates random 16 bytes and concatenates the received challenge to them – these result in 32 bytes of response: [RANDOM 16 bytes][CHALLENGE 16 bytes]
5. If Base 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
6. If it doesn't have pending messages in queue:
    a. Base sets SYNC bit in Header, and resets its internal TXsender to 1
7. Base now encrypts this and sends back to Server
8. *...data transfer...*
9. Server receives this message, verifies, decrypts and compares the received challenge value in last 16 bytes of decrypted message with the value originally sent to Base in Phase 1. If these match, the socket connection is authenticated
10. Server checks to see whether it must re-sync its copy of Base's sequence number to 0:
    a. If SYNC bit is set in received message, Server resets its copy of Base's TXsender to 0
    b. If SYNC bit is not set in received message, Server doesn't alter its copy of Base's TXsender but loads it from database to continue
11. Now Server prepares a reply for Base that will contain SYNC bit and TXserver (Server-Stored TXserver value)
12. Server now checks to see if it has some pending messages to send to Base
13. 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

```
14. If it doesn't have pending messages in queue:
       a. Base sets SYNC bit in Header, and resets its internal TXsender to 1
15. Server loads Server-Stored TXserver value and adds it as Data to this message
16. Server now sends this reply message to Server and in case it had pending
    messages queue it will also start sending those pending messages to Base
17. Base naturally receives this authentication reply message, treats the SYNC
    field accordingly and if it also had some pending messages in queue it will
    start sending them to Server
18. ...
```

*Pseudo-code*

If any time 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.

## 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 from Server -> 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": „babababababababababababababababa"
    }
}
```

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 from Client -> Server

#### *Pull unacknowledged messages*

Client can send a message to server that will force it to re-send all unacknowledged messages from its **txserver2client** queue. It should also be of type: **notification** but it is not obligatory.

Content of this message is (entire message example):

```
{
    „header": {
        „sync": false,
        „ack": false,
        „processed": false,
        „out_of_sync": false,
        „notification": true,
        „system_message": true,
        „backoff": false
```

```
        },
        „TXsender“: 0,
        „data“: {
                „type“: “pull_unacked”,
        }
    }
```

That’s it. After Server receives this system message it will start re-sending all unacknowledged items from its queue and hopefully Client will now receive them and acknowledge.

### Special System Messages sent from Server -> Base

Server will send a System Message to Base only when Base requests some special data from Server. This is currently implemented only for:

1. Read variable from Server
2. Get Timestamp from Server

Please read about these topics bellow.

### Special System Messages sent from Base -> Server

#### *Pull unacknowledged messages*

Base can send a message to server that will force it to re-send all unacknowledged messages from its **txserver2base** queue. It should also be of type: **notification** but it is not obligatory.

Content of this message is (without protocol header sections, only payload **data** section):

```
01
```

That’s it. After Server receives this system message it will start re-sending all unacknowledged items from its queue and hopefully Base will now receive them and acknowledge.

#### *Enable Keep-Alive on current socket connection*

In case Base’s network hardware replies to Keep-Alive messages, this option can be enabled by sending the appropriate system message.

Content of this message is (without protocol header sections, only payload **data** section):

```
02
```

If Base’s network hardware supports Keep-Alive messages, it is strongly recommended to enable this option since it helps connected Clients know when Base’s socket connection drops almost immediately.

#### *Disable Keep-Alive on current socket connection*

In case Base’s network hardware replies to Keep-Alive messages, this option can be enabled by sending the appropriate system message. **Note: Keep-Alive is disabled by default for Base Socket for each new connection.**

Content of this message is (without protocol header sections, only payload **data** section):

```
03
```

### *Save variable to Server*

It is possible to save 4-byte variables to Server's database and read them back at any time later on.

Content of this message is (without protocol header sections, only payload **data** section):

```
04 01020304 1A5BC104
```

Explanation of message parts:

- 0x04 = first byte is the command (always 0x04)
- 0x01020304 = variable ID (any 4 bytes)
- 0x1A5BC104 = variable VALUE (any 4 bytes)

### *Read variable from Server*

Reading previously stored 4-byte variables back from Server is done by requesting it using this message type.

Content of this message is (without protocol header sections, only payload **data** section):

```
05 01020304
```

Explanation of message parts:

- 0x05 = first byte is the command (always 0x04)
- 0x01020304 = variable ID (any 4 bytes)

The message will then be pushed to Base and will contain this format (without protocol header sections, only payload **data** section):

```
05 01020304 1A5BC104
```

Explanation of message parts:

- 0x04 = first byte is the command (always 0x04)
- 0x01020304 = variable ID (any 4 bytes)
- 0x1A5BC104 = variable VALUE (4 bytes from database)

In case Base requests a variable that currently does not exist in Server, it will return zero-value variable: 0x00000000.

### *Get Timestamp from Server*

Base can request a Timestamp from server by issuing this system command.

Content of this message is (without protocol header sections, only payload **data** section):

```
06
```

Server will then push the timestamp to Base with contents (without protocol header sections, only payload **data** section, in hexadecimal format):

`06 02000104 0102 0008 0108 0402 0503 01` = (2014/12/08 18:42:53 Monday)

## Document Change Log

| Date | Changed By | Summary |
|---|---|---|
| **2014-09-08** | Trax, trax@elektronika.ba | Initial document write |
| **2014-09-09** | Trax, trax@elektronika.ba | Still writing initial document... |
| **2014-09-19** | Trax, trax@elektronika.ba | Changed the way OUT-OF-SYNC situation is handled. Added Special System Message to turn on/off KEEP_ALIVE on server side. |
| **2014-09-28** | Trax, trax@elektronika.ba | Changing system to support many Bases to many Clients instead just one Base per many Clients. |
| **2014-09-29** | Trax, trax@elektronika.ba | Changed the way Clients login, instead of using username/password there is now **auth_token** parameter for Client authorization. This token is generated by the PHP admin web site. |
| **2014-10-01** | Trax, trax@elektronika.ba | Changed baseid type in JSON messages to array, so that server can accept a group of BaseIDs |
| **2014-12-07** | Trax, trax@elektronika.ba | Updating document for v1 of the system (added encryption of socket communication). |
| **2014-12-08** | Trax, trax@elektronika.ba | Completed the document with changes implemented. |
| | | |