# CTRL – IoT

## 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)*

**CTRL**  
Socket Server  
(NodeJS)

Hardware **Base**  
(Internet-connected hardware)

Software **Client**

*Binary Messages*

*JSON Messages*

**Internet**

Software **Client**

## Message Interchange Protocol v0.4

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 Interchange 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.

### Synchronization Mechanism

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:    1. 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    1. 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 a reply 6. ... 7. 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    1. If SYNC bit is set in authentication message:       1. Server resets its copy of Base’s TXsender to 0    2. If SYNC bit is not set in authentication message:       1. Server doesn’t alter its copy of Base’s TXsender 8. Server now checks to see if it has some pending messages to send to Base 9. Write me……… |

*Pseudo-code*

In case a de-synchronization situation happens, both parties should flush their message queue and re-connect.

### JSON Messages

This message type is a string of JSON with three objects: **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“: 501,  „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. *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.* 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.  *Currently not implemented in Client<->Server communication because we assume that both Server and Client have enough storage space and processing power.*  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.

#### 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

a

## Base

a

## Client

a

# Document Change Log

|  |  |  |
| --- | --- | --- |
| Date | Changed By | Summary |
| 2014-09-08 | Trax, trax@elektronika.ba | Initial document write |
|  |  |  |
|  |  |  |
|  |  |  |