|  |
| --- |
| Blue Husky Programming |
| BHIM Implementation Spec |
| How to implement Blue Husky’s Instant Messenger’s client and server in any high-level programming language |

|  |
| --- |
| Kyli Rouge  6-12-2014 |

Table of Contents

[2 Definitions 3](#_Toc390617275)

[2.1 Users 3](#_Toc390617276)

[2.1.1 Players 3](#_Toc390617277)

[2.1.2 Characters 3](#_Toc390617278)

[2.2 Chats 3](#_Toc390617279)

[2.3 Technology 3](#_Toc390617280)

[2.3.1 Timestamp 3](#_Toc390617281)

[2.3.2 IP Address 3](#_Toc390617282)

[2.4 Messages 3](#_Toc390617283)

[2.4.1 Communication Port 3](#_Toc390617284)

[2.4.2 Message Structure 5](#_Toc390617285)

[2.4.3 Message Types 7](#_Toc390617286)

[2.4.4 Roleplay Message 9](#_Toc390617287)

[2.4.5 Roleplay Message Parsing 9](#_Toc390617288)

[3 The Client 9](#_Toc390617289)

[3.1 Parsing User Input 9](#_Toc390617290)

[3.2 Sending Messages 9](#_Toc390617291)

[3.2.1 Compiling User Input into a Sendable Message 9](#_Toc390617292)

[3.2.2 Constructing a Special Message 9](#_Toc390617293)

[3.3 Receiving Messages 9](#_Toc390617294)

[3.3.1 Receiving a Special Message 10](#_Toc390617295)

[3.4 Maintaining a Local Database 12](#_Toc390617296)

[3.4.1 Mirror Database 12](#_Toc390617297)

[3.4.2 Conversation History 12](#_Toc390617298)

[4 The Server 12](#_Toc390617299)

[4.1 UUIDs 12](#_Toc390617300)

[4.1.1 Creation 12](#_Toc390617301)

[4.1.2 Storage 13](#_Toc390617302)

[4.1.3 Retrieval 13](#_Toc390617303)

[4.1.4 Maintenance 14](#_Toc390617304)

[4.2 User-created data 14](#_Toc390617305)

[4.2.1 Storage 14](#_Toc390617306)

[4.2.2 Retrieval 15](#_Toc390617307)

# The Short of It

BHIM is a peer-to-peer instant messenger which allows the users to send messages and media to each other, andor roleplay with each other either from their main account or through proxies called characters.

# Definitions

These are definitions essential to understand before implementing BHIM

## Users

A user is an individual human who uses BHIM. They are considered a user as long as they maintain their contact information on the server

### Players

“Player” is the term used for the physical user who uses BHIM. Each player has a profile that represents the player, emself.

### Characters

“Character” is the term used for a virtual profile owned and operated by a player. One player may have many characters, but each character can only be controlled by one player.

### Contacts

“Contact” is a general term used for anything that can be kept in a contact list. I.E. players, characters, and roleplays.

## Chats

A chat is any forum of communication between one or more users. There are two distinct classes of chat:

1. **Private:** Any chat involving exactly two users, where neither user can leave, and no new user can join. Roleplay messages won’t be parsed in this type of chat.
2. **Roleplay:** Any chat involving one or more users, where any user can leave, and any user can be invited to join. Roleplay messages are always parsed in this type of chat.

## Technology

### Timestamp

A BHIM timestamp is defined as a 64-bit integer denoting the date and time at which the message was sent as the number of milliseconds since January 1st, 1970 at 00:00:00.0000 (12:00:00.0000 AM)

### IP Address

1. BHIM clients and servers should not care what version of IP to use. Preferably, support for IPv4 and IPv6 should be built-in. If any new version emerges, or if compatibility with an old version is required, the client must be able to use these.

## Messages

A message is a string of text the user wishes to send to one or more other users and all the meta data that describes this interaction.

### Communication Port

BHIM messages are all sent and received through **port 3158**

BHIM uses port 3158 because that’s the standard IMPP port. Servers and clients must communicate through this port.

### Message Structure

This is the general structure of any message sent to or from a BHIM client

|  |  |  |
| --- | --- | --- |
| Section Name | Section Content | Content Description |
| Start of Header | 1 Byte: U+0001 (SOH) | The message and all its header data starts here |
| [Message Type Code](#_Message_Types) | 1 Byte: 8-bit integer | This is the type of message to be sent. For instance, it can be plain text, action, mix, et cetera. |
| Futureproofing | 2 Bytes | Reserved, yet undefined bytes for future use |
| Message Sender | 8 Bytes: 64-bit integer | The sending user’s UUID |
| Message Recipient | 8 Bytes: 64-bit integer | The recipient’s UUID |
| Time of Message Sending | 8 Bytes: 64-bit integer | A BHIM timestamp |
| Message Body | Arbitrary-length Unicode text | This is the message text that is sent. It is an arbitrary-length string of Unicode characters.    See the "[Message Types](#_Message_Types)" section for what this might contain. |
| Checksum | 4 Bytes: 32-bit integer | A checksum of the entire preceding message (Start of Header, Message Sender, Recipient, Type Code, Time of Sending, and Body. Note this does not include this checksum or the following EOT character) as an MD5 hash. |
| End of Transmission | 1 Byte: U+0004 (EOT) | The entire message is complete. Parse and display it to the recipient. Any data following this character is invalid and should be ignored. |

#### Message Structure Diagram

| **Offsets** | **Octet** | **0** | | | | | | | | **1** | | | | | | | | **2** | | | | | | | | **3** | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Octet** | **Bit** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **0** | **0** | Start Byte | | | | | | | | Type Code | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | **32** | Message Sender UUID | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **8** | **64** |
| **12** | **96** |
| **16** | **128** |
| **20** | **160** | Message Recipient UUID | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **24** | **192** |
| **28** | **224** |
| **32** | **256** |
| **36** | **288** | Time of Message Sending | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **40** | **320** |
| **44** | **352** |
| **48** | **384** |
| **52** | **416** | Message Body | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **⋮** | **⋮** |
| **len – 16** | **len – 64** | Checksum | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **len – 8** | **len – 32** |
| **len** | **len** | End Byte | | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

This makes plain the **448-bit (56-byte) header** and **72-bit (9-byte) footer**.

### Message Types

|  |  |  |  |
| --- | --- | --- | --- |
| Type Name | Type Code | Type Description | Body Content |
| Plain text | **00000000**  **0x00** | A message to another user or chatroom | Arbitrary-length Unicode string with arbitrary content |
| Media | **00000001**  **0x01** | An alert that there is a large packet of data to be transmitted only when both sender and receiver agree a transmission should take place | FTP URL pointing to the file. Note that this can be hosted by the Client on a temporary FTP server, or uploaded to an external, 3rd-party server. |
| Request for message change | **00000010**  **0x02** | A request that a previously-sent message should be edited or removed entirely | The MD5 Checksum of the message to be changed immediately followed by the new text of the message, or nothing if the message should be deleted |
| Announcement of information update | **00010000**  **0x10** | An announcement that the user’s meta information has changed | JSON indicating what information has been updated |
| Request for information update | **00010001**  **0x11** | A request that another user’s or chatroom’s meta info be sent back to the user for purposes of updating the local contact list | JSON indicating what information should be sent back. Response messages should adhere to the same structure. |
| Announcement of successful message receipt | **00010010**  **0x12** | A message was successfully received | The MD5 checksum of the received message |
| Request for re-send | **10000000**  **0x80** | A request that a specific message should be re-sent | The MD5 checksum of the message to be re-sent |
| Sending failed | **10000001**  **0x81** | An announcement that a message could not be sent | The MD5 checksum of the failed message |
| Sending failed permanently | **10000010**  **0x82** | An announcement that a message could not be sent and re-sending is impossible | The MD5 checksum of the permanently failed message |

Type Codes are structured so the broad categories (User-initiated, automated info, and automated error) are in the first nibble, and the sub-categories are in the second nibble.

### Roleplay Message

A roleplay message is a message the user intends to be placed in the context of a roleplay. This is defined as:

1. Any message sent in a roleplay chat
2. Any message sent in a private chat that has formatting described in the roleplay message formatting rules

There are three distinct voices in a roleplay move:

1. Speech: a character talking
2. Action or narration: description of a scene or action in the scene
3. Out-of-character speech: a user talking as emself

# The Client

## Parsing User Input

If the message is not a roleplay message, any below steps involving message parsing rules should be skipped.

### Roleplay Message Parsing Rules

A roleplay message parsing rule is any rule the user has specified as how their roleplay move should look. BHIM considers the following to be roleplay moves:

* Plain text sent in a roleplay chat
* Text surrounded by predefined delimiters
* Text preceded by a predefined delimiter
* Text surrounded by predefined delimiters after a predefined delimiter, with optional text between the preceding delimiter and the surrounding delimiters

A parsing rule must be able to:

* Return a Boolean indicating if the user’s input matches the rule
* Provide rules for converting the user’s input into a machine-readable format

### Roleplay Message Parsing

A message is considered unparsed when a user has typed it and requested it be sent. It may contain any Unicode character, except control characters. The following are the stages of **how to handle an unparsed message**:

1. If this is an automated message, skip to step 5
2. Any control characters (U+0000 to U+001F) in this string must be immediately removed
3. Any < (U+003C Less-than sign) or > (U+003E Greater-than sign) must be replaced with an HTML escape sequence (preferably &lt; and &gt;, respectively)
4. Conditional Parsing
   1. If no rules are matched or only unformatted text rules are matched, perform a simple format:
      * If a speech rule matches, do nothing
      * If a move or OOC rule matches, surround the entire string in the generic XML
   2. If the message matches any parsing rule for roleplay messages, apply those rules and format the message into generic XML. See [Parsing User Input](#_Parsing_User_Input) for detailed instructions on how to accomplish this.

## Sending Messages

### Compiling User Input into a Sendable Message

(list continued from [Roleplay Message Parsing](#_Roleplay_Message_Parsing))

1. Construct the message
   1. Save the current time as a 64-bit integer representing the number of milliseconds from 00:00:00.0000 on January 1st, 1970
   2. Start a character sequence, preferably one designed to be used in a loop
   3. Append the Unicode start-of-header character (U+0001 SOH: 00000001) to the beginning
   4. Detect and append the message’s type code
   5. Append two bytes of data. The value of these doesn’t matter, so we suggest you simply use 00000000 00000000
   6. Fetch and append the sending user’s UUID
   7. Fetch and append the recipient user’s UUID
   8. Fetch and append the previously-calculated timestamp
   9. Append the message body
   10. Pass the string so far through an MD5 hashing system and append the resulting hash
   11. Append the Unicode end-of-transmission character (U+0004 EOT: 00000100) to the end
2. Send the message
3. Ask [the local database](#_Maintaining_a_Local) to what IPs to send the message
4. If the local database returns none or an error, ask the external database to what IPs to send the message
5. Ensure no duplicate IPs in this list
6. Send the message to all these IPs over a TCP connection through [the BHIM port](#_Communication_Port)

### Constructing a Special Message

A special message is anything that’s not text the user typed and wishes to send

1. Media Message
   1. Either set up an FTP server that will be able to send the file once the recipient(s) request it, or upload the file to an external FTP server that can relay the media to the recipient(s)
   2. Construct a valid URL to the file on the FTP server and use that as the entire message body
2. Request for Message Change
   1. Start the message body with the timestamp and checksum of the message to be deleted
   2. If the user wishes to edit the message, follow this with the new text of the message. Else, if the user wishes to delete the message, do not follow the timestamp and checksum with anything.
3. Announcement of information update
   1. Assemble all the information the user wishes to refresh in a JSON string. Below is an example with whitespace added (the sent message should exclude unnecessary whitespace):  
      {  
       "<UUID>": [  
       "displayname",  
       "icon",  
       "tagline"  
       ]  
      }
   2. In the above example, replace <UUID> with the recipient’s UUID

## Receiving Messages

A dedicated server should be constantly listening for incoming messages on [the BHIM port](#_Communication_Port). Upon receiving a message, the client should do the following:

1. Prepare the following variables:
   1. two 1-byte integers
   2. one 2-byte integer
   3. one 4-byte integer
   4. three 8-byte integers
   5. a character sequencer, preferably designed to be used in a loop
2. Start receiving bytes
   1. Ensure the first byte is U+0001 SOH (00000001)
      * If it is not, discard the message as malformed and do not send a reply
   2. Save the next byte as the message type
   3. Save the next two bytes as the futureproofing bytes, but no not analyze them
   4. Save the next 8 bytes as the sender’s UUID
   5. Save the next 8 bytes as the recipient’s UUID
   6. Save the next 8 bytes as the timestamp of when the message was sent
   7. Append all the following bytes as the message body into the character sequence
   8. Once the end byte (U+0004 EOT (00000100)) is received, close the connection and discard any further data
   9. Remove from (or do not append to) the message body the EOT byte
   10. Save the 4 bytes before EOT as the checksum and remove from (or do not append to) the message body these 4 bytes
3. Check the message
   1. Create a character sequence comprised of the start byte, type byte, futureproofing bytes, sender UUID, recipient UUID, timestamp, and message body in that order with no delimiters
   2. Run this character sequence through an MD5 hashing system
   3. Ensure the resulting hash is exactly equal to the checksum sent with the message
   4. Ensure the recipient’s UUID is that of a profile owned by this player or a roleplay that this player is in
4. Respond appropriately
   1. If any step in checking failed, or if the EOT byte was never received before a timeout, send a “[Request for re-send](#_Message_Types_1)” message
   2. If the message was successfully received, send an “[Announcement of successful message receipt”](#_Message_Types_1) message
   3. If the type was “Request for information update”, respond with the appropriate information as described in the “[Constructing a Special Message](#_Constructing_a_Special)” section

### Receiving a Special Message

1. If this is not a text message:
   * Media
     1. Parse the entire message body as a single URL
     2. Ensure it’s a valid URL
     3. Ensure its protocol is FTP
     4. Alert the user that the sender wishes to send a file
     5. Iff the user agrees to download the file:
        1. Connect to the FTP server
        2. Request the file specified at the URL
        3. Download the response to the user-specified location, or a predetermined one if none specified (Preferably a system Downloads folder)
        4. Once the download is complete, provide a method of presenting the media to the user. E.G. an inline image, a link to the file, etc.
   * Request for message change
     1. Save the first 64 bits of the message body as the MD5 Checksum of the target message (the already-sent message that we are editing), then remove those bits from the saved new message body.
     2. If the user owns the UUID in the recipient field of the new message, save that as the UUID of the sender of the target message. Else, stop here and deny the user the ability to edit in a visually or aurally apparent way.
     3. Find the target message by searching for all matching archived messages’ MD5 hashes
     4. If the new message body is the empty string, remove the target message from the conversation history. Else, replace the text of the target message with the body text of the new message.
   * Announcement of information update
     1. Parse the message body as a JavaScript object
     2. TODO: How to update the information in this JSON string
   * Request for information update
     1. Parse the message body as a JavaScript object
     2. TODO: How to fill its fields with relevant info
     3. Parse this JavaScript object into a JSON string
     4. Send an Announcement of Information Update message back to the sender
   * Announcement of successful message receipt
     1. Parse the new message body as a 64-bit integer representing the MD5 checksum of the target message
     2. Find the target message by searching for its MD5 checksum
     3. If found, mark the target message as received. Else, fail silently.
   * Request for re-send
     1. Parse the new message body as a 64-bit integer representing the MD5 checksum of the target message
     2. Find the target message by searching for its MD5 checksum
     3. If found, construct and send a message identical to the one used to send it at first (i.e. use the archived timestamp instead of a new one). Else, fail silently.
   * Sending failed
     1. Parse the new message body as a 64-bit integer representing the MD5 checksum of the target message
     2. Find the target message by searching for its MD5 checksum
     3. If found, send a request for re-send to the original message sender. Else, fail silently.
   * Sending failed permanently
     1. Parse the new message body as a 64-bit integer representing the MD5 checksum of the target message
     2. Find the target message by searching for its MD5 checksum
     3. If found, visually mark the target message as failed to send in a way that’s immediately apparent to the user. Else, fail silently.

## Maintaining a Local Database

Each BHIM client must maintain a local database of its contacts’ IPs and conversation histories

### Mirror Database

Each BHIM client must cache the IPs of its contacts, as told by the server. These must be added, modified, and removed as they change on the server.

After a set amount of time (preferably changeable by the user), these are assumed no longer accurate, and must be re-fetched from the server either automatically or when a message is attempted to be sent. If the server fails to respond before timeout, or responds with an empty list, this contact must be marked as offline.

### Conversation History

Every message sent and received must be archived with its MD5 checksum as the primary key and timestamp as the secondary key. That is to say, when a message is to be referenced directly, all entries with its MD5 hash are to be fetched. If more than one is fetched, then the first one with the same timestamp is to be selected.

Conversation history is to be kept forever unless specified by the user’s preferences.

# The Server

The Server for BHIM works like an HTTP nameserver.

The sole duty of the BHIM server is to maintain a list of UUIDs and IPs for all currently on-line BHIM clients. No other data is to be sent to, stored on, or sent from a BHIM server.

## UUIDs

The main role of the server is to create and maintain all UUIDs

As the name implies, a UUID is a perfectly unique way to identify a contact (player, character, or roleplay). They should be 64-bit integers for ease of storage, sorting, and retrieval

### Creation

A UUID must be created when:

1. A player first creates a login for BHIM
2. A player creates a new character
3. A player creates a new roleplay

Upon these, the client will send the following respective information to the server:

1. **Player:** The player’s username, IP, and MAC address
2. **Character:** The player’s UUID and the character’s name
3. **Roleplay:** Each participating player’s and character’s UUID

The server must use these to create a single UUID that the client can use to make requests about each contact. The client will POST to http://<server address>/BHIM/uuid the following information for the following contacts:

1. **Player:**
   1. Username: <username>
   2. IP: <IP address, any version>
   3. MAC: <MAC address>
2. **Character:**
   1. Character: <character name>
   2. User: <user UUID>
3. **Roleplay:**
   1. Participants: <non-delimited list of participant UUIDs>

In order **to create the UUIDs**, the server must do the following:

1. Concatenate all given POST values into one non-delimited string
2. Pass this string through an MD5 hashing system
3. Place the resulting hash into a **64-bit variable**.
   * Note that this means the last 64 bits are ignored. This is intended, as no one server should ever have more than 18 quintillion contacts.
4. Attempt to place this variable in a sorted database of all UUIDs. If another UUID of the same value exists, increment the value of this variable by 1. If that UUID is taken, decrement the value by 2. Repeat until a unique UUID is found. See the [Maintenance](#_Maintenance) section for how to prepare for a full database.

Once again, note that ***only the IPs and UUIDs of contacts should be stored in a BHIM server. No other data should be stored.***

### Storage

The UUIDs must be stored in a database that can also store its related IPs and other meta data. The specific implementation of this is not specified, but they must be easily and rapidly retrievable.

### Retrieval

To retrieve a contact’s UUID, the client will GET to http://<server address>/BHIM/uuid the following information for the following contacts:

1. **Player:**
   1. Type: P
   2. Username: <username>
   3. IP: <IP address, any version>
   4. MAC: <MAC address>
2. **Character:**
   1. Type: C
   2. Character: <character name>
   3. User: <user UUID>
3. **Roleplay:**
   1. Type: R
   2. Participants: <non-delimited list of participant UUIDs>

Now, the server must respond with ***only*** the UUID that exactly matches the given data

### Maintenance

1. A player’s UUID must be kept until that player explicitly requests it be removed.
2. A character’s UUID must be kept until the owning player requests it be removed.
3. A roleplay’s UUID must be kept until all participants have left the roleplay.
4. In the end times:
   * Upon *approaching* the event that all possible 64-bit UUIDs are used up, the contacts holding the most unused 1% should be alerted that their data will be removed from the server due to inactivity.
   * Upon *reaching* the event that all possible 64-bit UUIDs are used up, those of the aforementioned 1% who have still not used their profile are to be removed from the server
   * In every case, the new UUID should be immediately inserted.

## User-created data

### Storage

The server must **NOT** store any meta data that is **NOT** specified in this section

1. **Player:**
   1. Currently logged-in IP(s) (array of integers)
2. **Character:**
   1. Current IP(s) (array of integers)
3. **Roleplay:**
   1. IP(s) of all participants (array of integers)

The exact storage method is not specified. All that is required is expedience in delivering the requested content.

In the interest of abstraction, futureproofing, and language agnostics, all information will be submitted via an HTTP POST request. The server implementation may choose to require an HTTP login before certain information can be submitted.

To store a contact’s IPs, the client will POST to http://<server address>/BHIM/ip the following information for the following contacts:

1. **Player:**
   1. Type: P
   2. IPs: <comma-separated list of IP addresses>
2. **Character:**
   1. Type: C
   2. Character: <character name>
   3. User: <user UUID>
3. **Roleplay:**
   1. Type: R

Participants: <non-delimited list of participant UUIDs>

### Retrieval

Information must be retrieved via HTTP pages

In the interest of abstraction, futureproofing, and language agnostics, all information will be gathered via a webpage served over HTTP. The server implementation may choose to require an HTTP login before certain information can be retrieved.

A client may place any of the following keys in a GET request to http://<serveraddress>/BHIM/get?uuid=<player UUID> and receive the corresponding values in a JSON string:

1. **Player:**
   1. name: Display Name (character sequence)
   2. icon: Icon (FTP URL as a character sequence)
   3. tagline: Tagline (character sequence)
   4. ips: Currently logged-in IP(s) (array of integers)
   5. availability: Current availability (byte)
2. **Character:**
   1. name: Name (character sequence)
   2. icon: Icon (FTP URL as a character sequence)
   3. tagline: Tagline (character sequence)
   4. ips: Current IP(s) (array of integers)
   5. availability: Current availability (byte)
   6. traits: Traits (JSON character sequence)
   7. description: Description (character sequence)
3. **Roleplay:**
   1. name: Name (character sequence)
   2. icon: Icon (FTP URL as a character sequence)
   3. tagline: Tagline (character sequence)
   4. description: Description (character sequence)
   5. ips: All IP(s) of all participants (array of integers)