Packet White Paper: 251
Status: Informational
Author: Paula Dowie
Date: January 2025

JSON Mailbox Protocol (JMP)

Abstract

This document describes a modern, extensible protocol for machine to machine interactions between end-user clients and Packet Radio mail servers.

The protocol uses JSON, because it is relatively easy to generate and parse, is easily understood by humans, and is ideally suited to modern browser-based applications.

The protocol is intended as an easy retrofit to existing Amateur Packet Radio mailbox software.

Status of This Memo

This memo provides information for the Packet Radio community. It describes an existing protocol, and is aimed at Packet mailbox and client developers who wish to implement it. Distribution of this memo is unlimited.

Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Table of Contents

1. Introduction

   1.1. Reasons For Using JSON
   1.2. Disadvantage of JSON
   1.3. Connecting To A Mail Server
   1.4. Identifying the JSON Capability

2. Structure of Requests And Responses

3. Summary of Available Commands

4. Commands In Detail

4.1. Status Enquiry
4.2. Requesting a List of Messages
4.3. Requesting a Single Message
4.4. Posting a New Message
4.5. Deleting a Message
4.6. Disconnecting From The Mailbox

5. Notes

6. Security Considerations

7. Author's Address

8. References

1. Introduction

Since the earliest days of Amateur Packet Radio (a.k.a. "Packet"), mailboxes have had at least two "interfaces", one for use by humans and the other for automated "Store and Forward" operations. The former is generally intended for "real time" usage, and the latter is usually available only to other designated mailboxes.

Packet Radio is slow, with an average throughput often lower than the old 300/300 dial up modems. Users don't wish to sit around reading mail "live" over a radio link. Therefore, over the years there have arisen many client programs which automatically download and upload mail at a snail's pace, whilst at the same time presenting a fast, standardised user interface. The most famous of these was "Winpack", a more recent one being "Outpost".

A major problem with such clients is the wide variation in the user interface between Packet mailbox types. For example, when listing mail, the column layouts are non-standard, the column headings languages, pagination and commands differ, and some mailboxes use colour. These variations make it difficult for the client program to parse the mailbox output. This means that users often spend many hours "tweaking" their client programs for a particular server, only to have them break due to a minor change in the server software.

What Packet mailboxes sadly lack is a standardised machine-to-machine interface, intended for use over radio. Such interfaces do already exist, i.e. the WA7MBL and F6FBB mail exchange protocols. But they are for store and forward only, and available only to trusted peer mailboxes.

Some mailboxes have POP3, SMTP, NNTP and IMAP, but these are relatively complex to retrofit onto a Packet mailbox, and are not best suited to Packet.

This document outlines a novel machine-to-machine protocol for Packet Radio mail servers, intended for use by end-user client applications. The new protocol uses JSON [RFC8259] over a conventional Packet

"stream" connection. The protocol is available in XRouter.

1.1. Reasons For Using JSON

The protocol uses JSON, because JSON is:

- relatively easy to generate and parse
- designed for handling lists and complex structures
- ideally suited to modern browser-based applications.
- easily understood by humans, which aids debugging.
- extensible (new features can be easily added)

This protocol is not the "ultimate" solution, it is simply ONE possible solution. Much more efficient protocols could be designed, but JSON is ubiquitous nowadays. When the core protocol is JSON, it is ideally suited for use via HTTP, MQTT, WebSockets or plain text links. The same program code can, with a few tweaks, serve all these types of links.

Although JSON is verbose, it is of minor consequence, as the majority of the data transferred is the mail itself.

1.2. Disadvantage of JSON

A downside of JSON is that it is poor at handling binary data.

Certain characters such as newlines, quotes and backslashes must be "escaped" by preceding them with a backslash, thus doubling the space that they occupy. In addition, all other character codes below 32 must be escaped by converting them to six-character sequences. This considerably bloats binary data. Fortunately, packet mail rarely contains binary data.

If binary data proves to be a problem, it may be "wrapped" in a JSON friendly encapsulation, such as Base-64, for transmission between server and client.

1.3. Connecting To A Mail Server

This document does not address the means of connecting to a Packet mailbox. There are many ways to do so, such as:

- Connecting to a Packet node and issuing the 'BBS' command.
- Connecting to a specific "BBS-only" callsign on a node.
- Connecting to a "stand alone" BBS.
- Connecting via NetRomX [PWP109] to the BBS "service" on a node.
- Connecting via Telnet (amprnet or internet)

Of these, the only consistent one is the NetRomX service, because it was expressly designed for machine-to-machine interactions. The only problem with NetRomX is that, so far, it is only implemented by XRouter.

This protocol outlined in this document is intended to be a drop-in addition to existing mailbox software, not a radical redesign of packet mailbox architecture.

Therefore, at least for the time being, it is best if the connection to the mail server is manually scripted and remains outside the scope of this document.

1.4 Identifying the JMP Capability

One option would have been to add a new capability flag to the mailbox System Identifier (SID). However, the SID is intended for mailbox to mailbox operations, and is often not displayed to "user" connections. So that approach was not chosen.

Clients may verify the JMP capability by simply sending a JSON formatted command and observing the response. The "status" command is ideal for this. Mailboxes without the JMP capability will respond with a non-JSON error message.

2. Structure of Requests And Responses

The protocol is very simple. The client connects to a mailbox in the same way as any other "user", issues requests (commands), and receives responses. When it is finished, it disconnects. The connection persists throughout the session, but the operations themselves are stateless. It is the responsibility of the client to keep track of state if necessary.

All commands must be in JSON format, beginning with the opening curly bracket '{', and ending with the closing curly bracket '}'. This "object" must be followed by carriage return (i.e. '\r', 0x0d, ASCII 13), as per normal Packet Radio convention.

White space (spaces, tabs, newlines, etc.) is allowed within the JSON objects, but is not mandatory. Examples in this document may include white space for clarity.

Commands are not case sensitive, and the order of fields within command objects is unimportant.

All command objects must have a "cmd" field, otherwise the response is as follows:

```
{"error": "missing cmd"}
```

Response objects are also followed by carriage return.

If a command is not recognised, the response is:

```
{"error": "Unknown cmd"}
```

3. Summary of Available Commands

The command set is limited to the bare minimum required to list, read, send and delete mail. Other commands are available, but have been omitted for brevity.

The accepted values for the "cmd" field are as follows:

Command	Action
"status"	Returns basic info about the message base.
"list"	Lists mail headers
"read"	Reads a single message
"send"	Sends a message
"kill"	Deletes a message
"bye"	Disconnects the client from the mailbox.

It would have been theoretically possible to pare all request and response field names down to single characters, but the choice not to do so was deliberate. Little would have been gained in terms of bandwidth, at the expense of making the protocol hard for humans to read. A human-readable protocol is easier to debug, and there is less chance of programming errors.

4. Commands In Detail

This section describes each of the commands in more detail. Each command is fully described in its own subsection

4.1. Status Enquiry

The status enquiry acts as a quick verification that the mailbox is capable of operation via JSON, and returns some information about the message base.

The request is simply: {"cmd": "status"}

If the mailbox responds to JSON, it will reply with a JSON object something like this:

```
{"msgs": 23, "priv": 23, "bull": 0,
  "oldest": 1711381953, "newest": 1737831323}
```

4.2. Requesting a List of Messages

The "list" request has many options, and would usually be the first request in a download session. All fields except "cmd" are optional.

Field	Type	Description
"cmd"	string	"list" (mandatory)
"offset"	integer	Number of items to ignore, starting from most recent message, default 0
"maxitems"	integer	Maximum items to return, default 500
"before"	integer	Msgs rcvd before this UNIX time
"after"	integer	Msgs rcvd on or after this UNIX time
"reverse"	boolean	true=newest first, false=oldest first (default)
"type"	string	Type of message: ("P", "B" etc)
"status"	string	Message status: ($, H, F, R etc)
"to"	string	Destination callsign / bulletin topic
"at"	string	Destination host or distribution area
"from"	string	Sender's callsign
"subject"	string"	String to match in subject field

Example: {"cmd":"list", "type":"B", "to":"TECH", "maxitems":10}

Options, other than "offset" and "maxitems", act as filters, reducing the number of message items returned. For example, if the option "type":"B" is specified, only bulletins are returned.

If the request succeeds, the reply is an unnamed JSON object containing an array called "msgs". The array contains zero or more JSON objects, each representing one item of mail (message), and containing the following fields:

Name	Type	Description
"id"	integer	Message number.
"mid"	string	Message ID (MID or BID)
"rcvd"	integer	Date/time of message reception (*1)
"dated"	string	Human-readable version of above (*2)
"size"	integer	Length of the message body in bytes.
"type"	string	Type of message: (A, P, B, E, T etc)
"status"	string	Message status: (R, F, U etc) (*3)
"to"	string	Destination address (*4)
"from"	string	Callsign of the message's creator.
"subject"	string	Message subject (32 chars max)

(*1) in Unix time, i.e seconds since 1st Jan 1970 UTC
(*2) e.g. "2024-06-11T07:00:12Z"
(*3) type and status may in future be unambiguous words
(*4) e.g. "g8pzt", "all@gbr", "g8pzt@gb7pzt.#24.gbr.eu"

For each mail item, only its header information is returned. Mail bodies must be requested individually (see below).

By default, messages are listed in NORMAL order, i.e. oldest first.

The list contains only those messages that the user is authorised to see, i.e. for non-sysops this means bulletins and private mail originated by, or addressed to, the user.

If there is not enough memory the satisfy the request, the response is: {"error": "2 (no memory)"}.

4.3. Requesting a Single Message

The "read" request is used to download messages, chosen from the response to a "list" request, from the server to the client. The request fields are as follows:

Field	Type	Description
"cmd"	string	"read" (mandatory)
"id"	integer	Message number (mandatory)
"hdrs"	integer	Routing hdrs (optional): 0=None, 1=first, 2=all

If the "hdrs" option is omitted, the default is to include no routing headers.

Example: {"cmd":"read", "id":2847, "hdrs":2}

If the request is successful, the reply is an unnamed JSON object containing at least the following fields:

Name	Type	Description
"id"	integer	Message number.
"mid"	string	Message ID (MID or BID)
"rcvd"	integer	Date/time of message reception (*1).
"dated"	string	Human-readable version of above (*2)
"size"	integer	Length of the message body in bytes.
"type"	string	Type of message: (A, P, B, E, T etc)
"status"	string	Message status: (R, F, U etc) (*3)
"to"	string	Destination address (*4)
"from"	string	Callsign of the message's creator.
"subject"	string	Message subject (32 chars max)
"text"	string	Body of the message (*5)

(*1) in Unix time, i.e seconds since 1st Jan 1970 UTC
(*2) e.g. "2024-06-11T07:00:12Z"
(*3) type and status may in future be unambiguous words?
(*4) e.g. "g8pzt", "all@gbr", "g8pzt@gb7pzt.#24.gbr.eu"
(*5) Message body includes all RFC822 and requested routing headers

If the user does not have the necessary privileges to access the message, the response is: {"error": "6 (permission denied)"}.

If the requested message number is not found, the response is: {"error": "10 (not found)"}.

If there is not enough memory the satisfy the request, the response is: {"error": "2 (no memory)"}.

4.4. Posting a New Message

The "send" request is used to upload a message to the mailbox, for onward transmission. The request must contain ALL of the following fields:

Name	Type	Description
"cmd"	string	"send"
"from"	string	Callsign of sender
"to"	string	Destination (see below)
"type"	string	Only "P" or "B" at present
"subject"	string	Subject of message (32 chars max)
"text"	string	Body of the message

For private messages the destination may be just a callsign, or <callsign>@<hierarchical-address>. For bulletins it may be simply <topic> or <topic>@<distribution>. For email it must be <user>@<host>.

If the request is successful, the reply is an unnamed JSON object containing the results of the operation, for example:

```
{"resource": "message", "id": "1741", "action": "created"}
```

If any field in the JSON request is missing or malformed, the response is: {"error": "12 (bad argument)"}.

If the user does not have the necessary privileges to create the message, the response is: {"error": "6 (permission denied)"}.

If there is not enough memory the satisfy the request, the response is: {"error": "2 (no memory)"}.

This method could also be used for copying a message to a new recipient.

4.5. Deleting a Message

The "kill" request is used to delete messages from the server. Messages are deleted by number, one at a time. The request fields are as follows:

Field	Type	Description
"cmd"	string	"kill" (mandatory)
"id"	integer	Message number (mandatory)

Example: {"cmd": "kill", "id": "2437"}

If the request is successful, the reply is an unnamed JSON object containing the results of the operation, for example:

```
{"resource": "message", "id": "1741", "action": "deleted"}
```

If any field in the JSON request is missing or malformed, the response is: {"error": "12 (bad argument)"}.

If the requested message number is not found, the response is: {"error": "10 (not found)"}.

If the user does not have the necessary privileges to delete the message, the response is: {"error": "6 (permission denied)"}.

Future versions may allow multiple deletion.

4.6. Disconnecting From The Mailbox

The "bye" request gracefully disconnects the client from the server. No response is sent. The request is as follows:

```
{"cmd": "bye"}
```

The connection is then terminated by the server.

5. Notes

The client should preferably use the "rcvd" timestamp to keep track of which messages have been received. The mailbox may renumber the message base from time to time, hence the "id" field of a message is only trustworthy for short-term usage.

However, renumbering tends to take place in the middle of the night when no clients are connected, so the "id" field can be trusted within a session.

The protocol is deliberately "one message at a time" (a) to avoid hogging the radio channel, and (b) to reduce the load on the server which may have to cache the responses to many clients simultaneously.

6. Security Considerations

Security is the responsibility of the mailbox, not this protocol.

In XRouter, mail security is based upon the client's callsign, and any privileges associated with that callsign. Clients with "standard" privileges are not allowed to view or edit mail that doesn't belong to them, i.e. which is not addressed to or from their callsign. They are also not allowed to send mail which has a "from" field that differs from their login callsign.

The tradition on Packet Radio has been that people are who they say they are. With many eyes watching a radio channel, callsign piracy tends to be easily spotted, and isn't generally a problem.

However, in some cases there may be a need for callsign verification, for example by the use of "one time codes", CHAP, or password grid challenges. These methods are usually included in standard mailbox software, hence this protocol includes no additional security.

7. Author's Address

Paula Dowie (protocol author)
g8pzt[at]blueyonder.co.uk (replace '[at]' with '@')

8. References

[PWP109] Dowie, P., "NET/ROM Data Multiplexing", PWP 109, July 2001.

[RFC8259] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 8259, December 2017.