The MsgLen Packet Format

Johannes Willkomm

2024-11-11

Contents

1 The MsgLen Packet Format **1**
- 1.1 Header formats 2
- 1.2 The meta information 3
- 1.3 Meta headers 3
- 1.4 Meta control messages 3
- 1.5 Meta information caching 4
- 1.6 Current implementation 4

1 The MsgLen Packet Format

A new flexible and modern packet format is proposed. MsgLen defines three families of protocols designed with the following simple principles: The packet header should be easy to interpret, as short as possible, efficiently readable and extensible. This is achieved by packets that consist of three sections: *header*, *meta* information and *data*.

The first section, the header, consists of a fixed number of 8, 16, or 24 bytes depending on the protocol family. The header contains in addition just two numbers: the meta length and the data length, which are the lengths in bytes of the meta and the data sections of the package that follow. The meta length is recommended to be a multiple of 8 bytes. Thus only two reads are required for each packet: one of the fixed header lenght and another with the sum of the meta length and the data length. This makes the entire packet very easy and efficient to read from a net workstream and the data will be placed at a 8 byte boundary in memory.

While the data part contains the binary user data, the meta information is encoded with JSON or XML. The meta section can thus transmit arbitrary structured information. Note that by just sending header and meta and a zero-length data section, a kind of side channel can be established.

Also, the flexible nature of the protocol opens the way for all kinds of sub-protocols and quasi-standards. For example, archives and structured messages may be constructed where the main MsgLen package contains inforation about the archive or message contents in the meta section.

JSON data in the meta section can be converted to XML on the fly using the **json2xml** function from the astunparse package. The meta information may also be compressed by one of the following compression algorithms: gzip, xz, bzip, or brötli.

1.1 Header formats

The family of MsgLen protocols is defined by its headers, of which there are three:

Name	Magic	Flags	MetaLength	HeaderLength	Total
mx	m [xh]	1 B	2 B	3 B	8 B
msgl	m s g [lbhd]	4 B	4 B	4 B	16 B
Msgl	M s g [lbhd]	4 B	8 B	8 B	24 B

The *mx* protocol family has just two members, one binary format and one ASCII variant:

Name	Format	0-1	2	3-4	5-7
mx	binary	x m	8 bit	16 bit	24 bit
mh	ASCII hex.	x h	4 bit	8 bit	12 bit

This provides 8 bits for flags, usually all zeros, 16 bits for the meta length and 24 bits for the data length.

The *msgl* protocol family has one binary format and three variants which result in a pure ASCII header. It provides in the binary format 32 bits for flags, 32 bits for the meta length and 32 bits for the data length:

Name	Format	0-3	4-7	8-11	12-15
		Magic	Meta-Length	Data-Length	Flags
msgl	binary	m s g l	32 bit	32 bit	32 bit
msgb	ASCII base64	m s g b	24 bit	24 bit	24 bit
msgh	ASCII hex.	m s g h	16 bit	16 bit	16 bit
msgd	ASCII dec.	m s g d	0-9999	0-9999	13 bit

The *Msgl* protocol family has one binary format and three variants with a pure ASCII header. It provides in the binary format 32 bits for flags, 64 bits for the meta length and 64 bits for the data length.

Name	Format	0-3	4-7	8-15	16-23
		Magic	Flags	Meta-Length	Data-Length
Msgl	binary	M s g l	32 bit	64 bit	64 bit
Msgb	ASCII base64	M s g b	24 bit	48 bit	48 bit
Msgh	ASCII hex.	M s g h	16 bit	32 bit	32 bit
Msgd	ASCII dec.	M s g d	13 bit	0-99999999	0-99999999

Currently there are no flags specified, they are usually all zeros. The hope is all of the requirements regarding the transport and protocol can be placed in the meta information. When flags are added in the future, the most important flags should be placed in the lower bits.

In the case of the ASCII formats, whitespace must be used for padding. Also, pure whitespace in the flags fields must be interpreted as the flags being all zero.

Note that, when the ASCII header forms are used, and the meta information is not compressed, and the data is UTF-8 encoded, then the entire packet is valid UTF-8 data.

An implementation may switch at any time between the members of one protocol family, but not to that of a different family unless explicitely requested by a suitable mechanism.

1.2 The meta information

The meta section of the MsgLen packet may contain JSON or XML data, optionally compressed, and padded to achieve a section length divisible by 8.

The meta section data must be whitespace padded when containing JSON or XML data and one of the ASCII header forms are used, or zero padded when it contains compressed binary data.

The meta section must be encoded using UTF-8.

1.3 Meta headers

Currently the following fields in the meta info are defined and used by the reference implementation:

encoding used to automatically encode and decode the data bytes from and to strings

pack python struct definition string, used to automatically pack and unpack binary data

We want to preliminarly reserve the meta data fields mentioned in the following sections, like protocol, error, warning, etc. While their implementation is not specified yet, the idea is that they carry the intended meaning.

We also soft-reserve all the names used by the HTTP protocol for its headers. When these are used by an implementation, they must be used in accordance with their intended meaning.

1.4 Meta control messages

Zero-length data packets may be used to signal state between the peers. The following fields may be used for such operations:

protocol mx, msgl, or Msgl

version

get-options respond by sending with the current meta state

reset-options clear local meta state

reset reset errors

restart restart service

flush

error

warning

msg

comment

pass list of fields that must be passed onwards

content-type unpack if known MIME type, such as text/json or text/xml

Implementations must signal error and warning states, and other typical notifications using the fields error, warning, msg, comment.

1.5 Meta information caching

Implementations should cache the meta information being send, by continuously performing a dictionary union of their current state of meta information and the meta information in the current package.

When this process would leave the implementation in an impossible state, it must not perform the option update and respond to the sender with an error message.

A service will provide the state meta state by sending the corresponding JSON when it receives a message containing the get-options field.

The reset-options field means that the receiver reset its local meta state to the empty dictionary.

1.6 Current implementation

The current implementation in this package is incomplete and uses an altogether different format:

Name	Format	0-3	4-7	8-11
		Magic	Meta-Length	Data-Length
msgl	binary	m s g l	32 bit	32 bit
msgb	ASCII base64	m s g b	24 bit	24 bit
msgh	ASCII hex.	m s g h	16 bit	16 bit.
msgd	ASCII dec.	m s g d	0-9999	0-9999

The implementation and this spec are supposed to merge in the future but both are obviously still subject to change.

The file msglen/msglen.py contains the core implementation, which constists of the classes `MsglenL`, `MsglenB`, `MsglenH`, and `MsglenD`. The first is the most extensive, the others only need to overwrite to methods: `_unpackHeader` and `packHeader`.

The classes can be given an open file object and then the methods readData, readMeta, and readHeader can be used to read the (binary) section data.

The methods `pack` can be used to construct an entire packet out of meta data and data. The methods `packHeader` and `metaHeader` can be used to construct the header and meta section byte data individually.

The method `unpackHeader` returns the triple (id, mlen, dlen) from 12 bytes of byte data. The combined meta and data section can be unpacked with the method `unpack`.

The method `reader` returns an asynchronous reader handle when passed an asyncio stream reader. The methods `writer` returns a regular function that writes data to a given stream with given meta dict and `packer` returns a function that packs data with a given meta dict.