Compact Message Exchange Protocol

1. Introduction

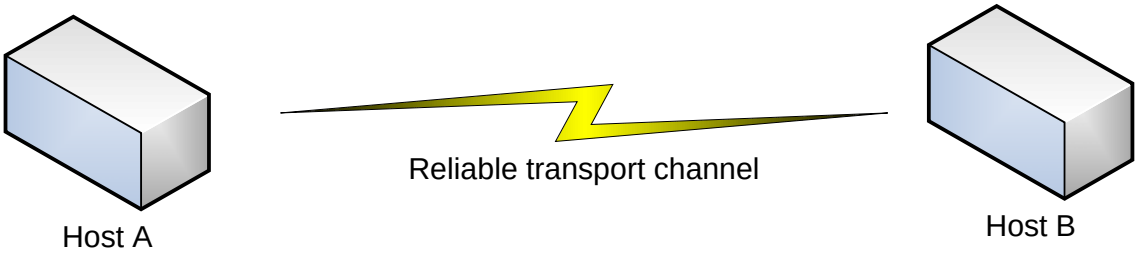
The objective of the Compact Message Exchange Protocol is to efficiently exchange data structures between entities.

The protocol is independent of any particular transmission subsystem and requires only an underlying reliable ordered data stream channel[1], such as TCP or RFCOMM.

Special attention has been given to the issue of minimization of protocol overhead and at the same time maintaining clear-text readability of the protocol. Because of this the construction of the protocol is a balance between a low-bandwidth approach and ease of debugging.

2. Communication model

2.1. Peer-to-peer communication



The protocol employs a peer-to-peer communication model. In this model, both connected parties are equivalent in their status, i.e. there is no distinction in the protocol itself between a server side and a client side. This distinction might be implied by the actual types of the parties; for example a mobile phone will act as a client when communicating with a central server.

2.2. Terminology

This document uses the Augmented Backus-Naur Form for expressing protocol constructs, as defined in RFC 4234[2]. We also define some basic terms in the BNF notation:

```
lchar = %x00-09 / %x0B-xFF          ; line-char

dchar = ALPHA / DIGIT               ; alfanum

nchar = ALPHA / DIGIT / '.' /
        '_' / '-'                   ; name-char
```

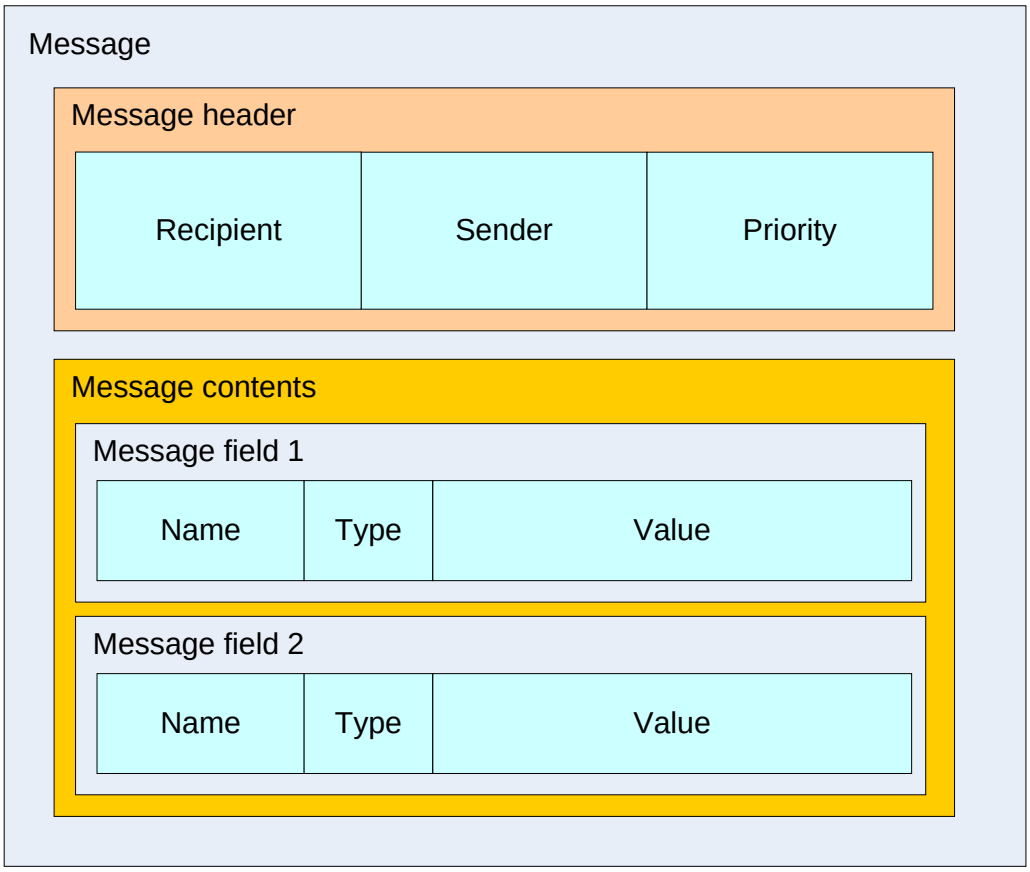
[1] "RFC 2821: Simple Mail Transfer Protocol", April 2001
[2] „RFC 4234: Augmented BNF for Syntax Specifications: ABNF", October 2005

2.2.1. Message object

A message object is a basic form of exchanging user data payload between peer entities defined in this protocol. A message object is a collection of fields. Each field has three associated attributes:

- field name – used to uniquely identify the field in the message,
- field type – used to specify the type of payload carried by the field,
- field value – contains the user data payload.

Apart from containing user data fields, a message object also carries information about its sender and the desired recipient. Both of these fields represent entities, such as class methods, procedures or subroutines, at the transmitting parties. That is, a message communication takes place between certain pieces of application code.



The above figure presents a structure of simple, two-field message object. The meaning of the Priority field is described in the *MSG command* section.

3. Protocol specification

3.1. The protocol line

The basic transmission unit that is considered as a whole is the protocol line:

```
line    = *lchar LF
```

meaning a line of ASCII characters ranging from 0 to 255 (except 10) terminated with an ASCII 10 (linefeed) character.

The communicating parties can thus buffer an incoming stream of bytes until a new-line character is received and then pass it on to further processing.

There are two types of protocol lines that are defined in the protocol. The first one is a command line (see *Command syntax*), which is used to indicate a request to execute a given command on the other end. The second one is a message line (see *MSG command*), which carries user data associated with a certain message object. The definition of a protocol line can be thus presented in this form:

```
line    = command_line / message_line
```

3.2. Protocol commands

3.2.1. Command syntax

Each command has the following syntax:

```
command_line = command_id [SP parameters] LF

command_id = 3*ALPHA
```

3.2.2. HLO command

The HLO command is used to initiate a new session. It is used by both parties to show that they can speak the CMEP protocol. The structure of the command is as follows:

```
command_id        = “HLO”
parameters        = client_name ‘/’ client_version SP
                    client_midp

client_name       = *nchar
client_version    = *dchar
client_midp       = *VCHAR
```

The *client_name* field is used to identify the client software (e.g. “wavu”), *client_version* identifies the version of the software (e.g. “1.0”) and *client_midp* can be used to convey information about the mobile clients capabilities.

An example HLO command can look like this:

```
HLO wavu/1.0 MIDP2 Bluetooth
```

3.2.3. MSG command

The MSG command is used to indicate the start of a transfer of a message object to a certain receiving entity at the other end. It is the main protocol command for transfer of user data. It has the following syntax:

```
command_id        = “MSG”
parameters        = recipient SP sender SP priority
```

```
recipient          = *nchar
sender             = *nchar
priority           = DIGIT
```

The *recipient* field identifies the entity that is to receive the message. This can be a name of a class and a method to execute on a server, a unique identifier for an instantiated class, or some other means of identifying the receiver. The *sender* field contains similar data but identifying the sender entity of the message. The *priority* field is used to indicate the transmission priority of this message and at the same time uniquely identifies all the parts that comprise the message.

An example MSG command can look like this:

```
MSG Security.Auth.login 3 1
```

The above command indicates that the sender will transmit a message with priority 1 from entity “3” to destination entity “Security.Auth.login”. The message body will follow this command.

3.2.3.1. Message body

The body of the message is comprised of multiple message lines:

```
body         = *message_line

message_line = priority specifier line_data LF

specifier    = ‘:’ / SP / ‘.’

line_data    = *lchar
```

There are three different types of message lines, that are distinguished by the *specifier* field. The methods for encoding a message object using these lines are discussed in *Message exchange*.

3.2.3.2. Field definition line

The field definition line is a message line whose specifier is set to „:” (a colon). This line is used to indicate the name of the field and the type of data that it contains. This line can also carry a single line user data payload.

The *line_data* for this field has the following format:

```
line_data    = field_name SP field_type [ ‘=’ field_value ]

field_name   = *nchar
field_type   = 1*3nchar
field_value  = *lchar
```

The *field_name* field specifies the name of the message field. The name can be empty, but otherwise the names SHOULD be unique within the message object. The *field_type* is a 3-

character string that identifies the type of data that is carried in this field. This information allows the communicating parties to properly interpret the data. For a list of defined field types, see *Field types* in the *Message exchange* section.

The *field_value* element, carrying the user data payload, is present only if the whole user data payload can be represented in a single line, i.e. it does not contain any linefeed characters. Otherwise this element is omitted, and the user data payload is transmitted using multiline data lines.

An example field definition line may look like this:

```
1:password str=my_password
```

In the above case, the field “password” contains only a single line of data, “my_password”. If a multiline payload was sent, the definition line could have looked like this:

```
1:password str
```

and the actual data would follow in multiline data lines.

3.2.3.3. Multiline data line

The multiline data line is a message line whose specifier is set to “ “ (a space). This line is used to transmit each line of the user data payload that is comprised of multiple lines (i.e. it contains at least one ASCII 10 – linefeed – character).

This line has the following format of *line_data*:

```
line_data    = *lchar
```

The above definition means that this line contains a single line of user data payload.

An example multiline data line could look like this:

```
1 this is a singleline of my data
```

3.2.3.4. Message end marker line

The message end marker line is used to indicate the end of the message object. Once the receiver encounters this line, it can pass the message buffer to a message decoder function. This type of message line is identified by a specifier field set to “.” (a dot).

This field has no meaningful *line_data* defined.

An example message end marker line would be:

```
1.
```

3.2.3.5. Peer response

After receiving a message object the receiver MUST send an ERR command to indicate whether the message was received correctly and got to its destination.

3.2.4. MSS command

The MSS command has the very same structure and meaning as the MSG command, that is it is used for transferring message objects, except for a single difference. The user data payload transmitted in an MSS message is encrypted.

Note: Only the user data payload undergoes encryption. None of the other protocol fields, like field name, field type, etc., are encrypted.

The encryption scheme is discussed in another section of this report.

3.2.5. ERR command

The ERR command is used to report a status code to the peer. The format of the command is as follows:

```
command_id   = "ERR"
parameters   = code recipient priority title

code         = 3DIGIT
recipient    = *nchar / '-'
priority     = DIGIT / '-'
title        = *VCHAR
```

The *code* entity indicates a status code returned by the peer. For a list of status codes, see *Status codes*. The *recipient* field can either contain the name of the entity at the other end that should receive this status message, or "-" (a dash) to indicate that the entity is unknown. The *priority* field indicates the message priority of the message that generated the status code, or a dash if the priority is unknown. The *title* field contains a descriptive message of the status code.

An example ERR command might look like this:

```
ERR 200 3 1 OK
```

4. The CMEP Procedures

4.1. Session initiation

To initiate a CMEP session, a party should first establish a reliable stream connection to a peer. Once such a communication channel is available, the party accepting the connection MUST send a HLO command. The initiating party MUST then respond with its own HLO command.

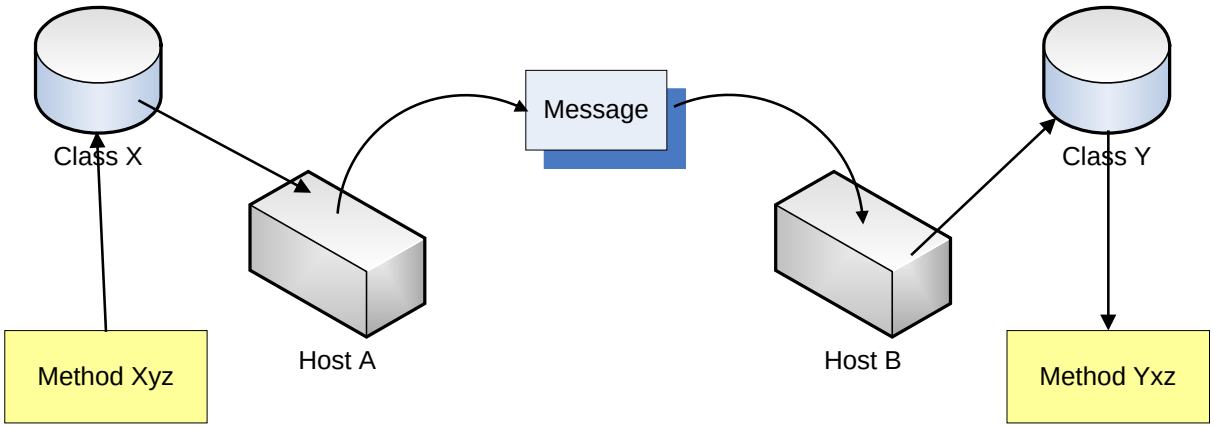
An example session initiation might look as follows (the ">" and "<" characters indicate the accepting and initiating parties, respectively):

```
> HLO server/1.1
< HLO wavu/1.0 MIDP 1.0 Bluetooth
```

After a session initiation completes, message exchange can then take place.

4.2. Message exchange

4.2.1. Exchange model



The message exchange model describes the procedure of sending a message object from one party to another. As messages are actually exchanged between pieces of application code (i.e. classes, procedures, etc.), the sender and recipient of a message are usually methods invoked at the respective ends.

The procedure for sending a message is as follows:

1. The sender creates a new empty message object and sets the message header fields (name of recipient, name of sender, message priority).
2. The sender adds any fields it wants to send in the message by encoding each field according the rules defined for the corresponding data type.
3. The sender put the message in the outgoing buffer for the given priority, line by line.
4. The sender periodically checks the outgoing buffer for different priorities and transmits a line out of each buffer in such a way, that the buffers with higher priorities will be given some preference.

The receiving end performs the following procedure:

1. Reads each incoming byte into an incoming buffer.
2. Once a linefeed character is encountered, the receiver checks for the type of line that was received:
    a. If the line conforms to the *message_line* specification of *field definition line* or *multiline data line*, the receiver strips off the priority digit and adds the remaining bytes to a incoming buffer corresponding to the message priority represented by the digit.
    b. If the line conforms to the *message_line* specification of *message end marker*, then the receiver should parse the whole incoming buffer for the associated *priority*.
    c. If the line conforms to the *command_line* specification, the receiver processes the given command. In particular, if the *command_id* is “MSG”, then the receiver should create an empty buffer for incoming data with the specified *priority*.

d. Otherwise, the receiver should discard the line and return a “400 Bad Request” status message.

4.2.2. Encoding

The protocol does not impose any special encoding rules for the transmitted user data payload, except for proper encoding of linefeed characters.

Once user data payload is represented by a certain field type, and thus encoded in the manner required by that field type (called the *field_payload*), it must undergo separation into individual message lines. This procedure can be carried out as follows:

1. If the *field_payload* does not contain any linefeed characters, it is represented as a single *field definition line* with *field_value* equal to *field_payload*, and the procedure terminates.
2. Otherwise, a *field definition line* without any *field_value* is outputted.
3. For each occurrence of a linefeed character in *field_payload*, a separate *multiline data line* is outputted containing a portion of the *field_payload* ranging from the byte following the previous occurrence (or from the beginning of *field_payload)* up to the current occurrence, but not including the linefeed character itself.
4. If *field_payload* ends with a linefeed character, then it is represented by an empty *multiline data line.*

To illustrate the above procedure consider the following two examples.

If the message of priority 2 is to contain a field “fullname” of type StringData with the following contents:

```
Smith, John T.
```

then it will be represented as a single *field definition line* in the message:

```
2:fullname str=Smith, John T.
```

If on the other hand, the same message is to contain a multiline field “address” of the type StringData with the following contents (note the empty line at the end):

```
46000 Center Oak Plaza
Sterling, VA 20166

```

it will be represented by the following message lines:

```
2:address str
2 46000 Center Oak Plaza
2 Sterling, VA 20166
2
2
```

4.2.3. Field types

Field types specify the data types that may be carried using the protocol and special type-specific encoding considerations. As defined in the *MSG command* section, each field type is

identified by a 3-character string and can carry an arbitrary stream of bytes (that do not, however, contain a linefeed character):

```
field_type   = 1*3nchar
field_value  = *lchar
```

4.2.3.1. StringData

The StringData type is used to carry unformed string information. There is no special encoding of the payload:

```
field_type   = “str”
field_payload= *OCTET
```

4.2.3.2. IntData

The IntData type is used to carry a signed integer value.

```
field_type   = “int”
field_payload= [ ‘-‘ ] *DIGIT
```

4.3. Returning status information

4.3.1. Informational 1xx

These codes are used to inform the other end about the current status of the program.

4.3.1.1. Keep-alive 100

This code is sent to the peer entity in order to test whether the other end is responding. The peer entity MUST respond with a “101 Alive” status code.

4.3.1.2. Alive 101

The reception of this status code informs the receiver that the other end is still capable of sending communication.

4.3.2. Successful 2xx

These codes are used to inform that a desired protocol operation was completed successfully.

4.3.2.1. OK 200

This status code informs the receiver that the last message was received properly by the other end.

4.3.3. Reserved 3xx

These status codes are reserved for future extensions to the protocol.

4.3.4. Peer Entity Error 4xx

These status codes are used to notify the peer entity that the data it sent was improper.

4.3.4.1. Bad Request 400

An unknown protocol command code was received or the protocol line did not conform to any known formats.

4.3.4.2. Malformed Message 401

A malformed message object was received.

4.3.4.3. Reserved 402

This status code is reserved for future use.

4.3.4.4. Reserved 403

This status code is reserved for future use.

4.3.4.5. Module Not Found 404

An unknown module was requested in the Recipient field of the message object.

4.3.4.6. Method Not Allowed 405

The Recipient field of the message object identifies a proper module, but this module does not allow the given method to be executed.

4.3.4.7. Session Uninitiated 406

The session initiation procedure was not completed properly.

4.3.5. Internal Errors 5xx

These status codes are used to notify the peer entity about a problem at the receiver's side that was not caused by improper data.

4.3.5.1. Internal Client Error 500

The application experienced an unidentifiable error.

5. References

1. "RFC 2821: Simple Mail Transfer Protocol", April 2001
2. „RFC 4234: Augmented BNF for Syntax Specifications: ABNF", October 2005