XEP-0426: Character counting in message bodies

Marvin Wißfeld
mailto:xmpp@larma.de
xmpp:jabber@larma.de

2022-12-27
Version 0.3.0

Status	Type	Short Name
Experimental	Informational	charcount

This document describes how to correctly count characters in message bodies. This is required when referencing a position in the body.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

Various use-cases require the possibility to reference a part of the message body or a specific position in it. This was realized by providing offsets from the beginning of the message (when referencing a region, those offsets would define begin and end of a region). XEPs doing so include In-Band Real Time Text (XEP-0301) [1], References (XEP-0372) [2] (and thereof derived Stateless Inline Media Sharing (XEP-0385) [3]) and Message Markup (XEP-0394) [4].
For these use-cases, it is highly relevant to decide how to count "characters" in a message body. While it at first sounds trivial, there are various ways of doing so in modern font systems. The purpose of this XEP is to define how characters shall be counted for the purpose of the aforementioned XEPs and any future XEP relying on a similar feature.

2 Character counting

When counting characters in a body, they shall be counted by their **number of Unicode code points**. Message bodies must be used as strings of the XML characters (as defined in §2.2 of XML 1.0 [5]). This means that, i.e. no Unicode normalization may be performed before determining offsets when receiving or after determining offsets when sending. Any kind of further body processing shall be performed after counting (e.g. /me·[6] as described in The /me Command (XEP-0245) [7] is always counted as 4 characters without considering the sending user's name). All references (as defined in §4.1 of XML 1.0 [8]) must be counted by their referenced character(s) and not the reference characters (e.g. the encoded & is counted as one decoded character &).

String	Grapheme cluster	UTF-8 bytes	UTF-16 units (2 bytes)	Code points
Hello, world!	13	13	13	13
You & Me	8	8	8	8
□□□□□□□	7	21	7	7

[1] XEP-0301: In-Band Real Time Text <https://xmpp.org/extensions/xep-0301.html>.
[2] XEP-0372: References <https://xmpp.org/extensions/xep-0372.html>.
[3] XEP-0385: Stateless Inline Media Sharing (SIMS) <https://xmpp.org/extensions/xep-0385.html>.
[4] XEP-0394: Message Markup <https://xmpp.org/extensions/xep-0394.html>.
[5] Extensible Markup Language (XML) 1.0 (Fourth Edition) <http://www.w3.org/TR/REC-xml/>.
[6] The middle dot is used to represent a space character and is not meant to be taken verbatim.
[7] XEP-0245: The /me Command <https://xmpp.org/extensions/xep-0245.html>.
[8] Extensible Markup Language (XML) 1.0 (Fourth Edition) <http://www.w3.org/TR/REC-xml/>.

String	Grapheme cluster	UTF-8 bytes	UTF-16 units (2 bytes)	Code points
□□ □□□□ □□	5 There are spaces between the emojis. You may also perceive this as more than 5 glyphs if your font or display engine does not support the required Unicode version.	43	21	13

2.1 Illegal offsets

As grapheme clusters may consist of multiple code points, a code point offset might be illegal if it points inside a grapheme cluster.
However, receiving entities SHOULD NOT consider illegal offsets invalid, as different Unicode versions may have different understanding of what a grapheme cluster is. Instead, receiving entities may choose one of the following behaviors:

- Split the grapheme cluster into multiple graphemes. In most cases, this is closest to the intended behavior. Many font display engines will do this automatically as needed.
- When the offset defines the end of a region, include the full grapheme cluster in the region. Otherwise, take the offset as if it pointed to the beginning of the grapheme cluster.

2.2 Developer notes

Some programming languages include a string type that operates directly on Unicode code points. If these types are used, offset numbers can be used as-is in string operations. Popular examples of such programming languages are Python and Haskell.
Other programming languages include a string type that operates on UTF-16 units. As can be seen in the table above, those match the number of code points in many cases and thus are sometimes confused to be the same. Popular examples of such programming languages are C#, Java and JavaScript.
C/C++ includes a wide character and string type. Those behave differently across platforms and as such should be used with care.

2.3 Rationale

The most obvious way of counting characters is to count them how humans would. This sounds easy when only having western scripts in mind but becomes more complicated in other scripts and most importantly is not well-defined across Unicode versions. New unicode versions regularly added new possibilities to build grapheme clusters, including from existing code points. To be forward compatible, counting grapheme clusters, graphemes, glyphs or similar is thus not an option. This leaves basically the two options of using the number of code units of the encoded string or the number of code points.
The main advantage of using the code units would be that those are native to many programming languages, easing the task for developers. However programming languages do not share a common encoding for their string type (C/C++ use UTF-8, C#/Java use UTF-16, Python 3 hides the internal encoding from the developer and only presents it in code points), so there is no best pick here. If one was to choose an encoding, the best choice would be UTF-8, the native encoding of XMPP. However this makes counting bytes a more complex task for programming languages that use a different encoding like UTF-16, as strings would need to be transcoded first.
Counting code points has the advantage that offset counts cannot point inside a code point. This could happen when using code units of any encoding that may use more than one unit to represent a code point (such as UTF-8 and UTF-16). If an offset count points inside a code point, that would be an invalid offset, raising more uncertainty of the correct behavior in such cases. Most notably the opportunity of splitting (as it exists for grapheme cluster) is not an option in that case, because splitting a code point would not create any usable output. Counting code points is widely supported in programming languages and can easily be implemented for encoded strings when not. The XML 1.0 [9] standard also defines a character as a unicode code point, thus counting code points is equivalent to counting XML characters.

3 Subsequences

When referencing a subsequence of the characters of a message body, the begin and end of the subsequence should be provided by two numbers, denoting the number of characters (counted as described above) before the begin of the subsequence or before the end of the subsequence, respectively. In other words, the begin is the index of the first character in the subsequence and the end is the index following the last character in the subsequence. That means, if a subsequence covers the full body, its begin should be given as 0 and its end should be given as the number of characters in the body.

[9] Extensible Markup Language (XML) 1.0 (Fourth Edition) <http://www.w3.org/TR/REC-xml/>.

3.1 Developer notes

Subsequence indexing in various programming languages match the convention described here. When using Python, the subsequence created by body[begin:end] matches all requirements of this document.
Some programming languages define subsequences by offset and length. In this case, begin matchs the offset while end-begin matches the length.

3.2 Rationale

The convention for subsequences was choosen because it has three main advantages: It matches subsequence indexing in various programming languages, end minus begin of a subsequence equal the length of the subsequence and the end of the first of two adjacent subsequence matches the begin of the second one.

4 Glossary

Unicode terminology used across this document, can be looked up in the Unicode glossary at https://www.unicode.org/glossary/.

5 IANA Considerations

This document requires no interaction with the Internet Assigned Numbers Authority (IANA) [10].

6 XMPP Registrar Considerations

This document requires no interaction with XMPP Registrar [11].

[10] The Internet Assigned Numbers Authority (IANA) is the central coordinator for the assignment of unique parameter values for Internet protocols, such as port numbers and URI schemes. For further information, see <http://www.iana.org/>.

[11] The XMPP Registrar maintains a list of reserved protocol namespaces as well as registries of parameters used in the context of XMPP extension protocols approved by the XMPP Standards Foundation. For further information, see <https://xmpp.org/registrar/>.

7 Acknowledgements

The author would like to thank Guus der Kinderen, Ralph Meijer, Jonas Schäfer, Lance Stout and others that provided feedback.