Network Working Group
Request for Comments: 1143

D. Bernstein
NYU

February 1990

The Q Method of Implementing TELNET Option Negotiation

Status of This Memo

This is RFC discusses an implementation approach to option negotiation in the Telnet protocol (RFC 854). It does not propose any changes to the TELNET protocol. Rather, it discusses the implementation of the protocol of one feature, only. This is not a protocol specification. This is an experimental method of implementing a protocol. This memo is not a recommendation of the Telnet Working Group of the Internet Engineering Task Force (IETF).

1. Introduction

This RFC amplifies, supplements, and extends the RFC 854 [7] option negotiation rules and guidelines, which are insufficient to prevent all option negotiation loops. This RFC also presents an example of correct implementation.

DISCUSSION:

The two items in this RFC of the most interest to implementors are 1. the examples of option negotiation loops given below; and 2. the example of a TELNET state machine preventing loops.

1. Implementors of TELNET should read the examples of option negotiation loops and beware that preventing such loops is a nontrivial task.

2. Section 7 of this RFC shows by example a working method of avoiding loops. It prescribes the state information that you must keep about each side of each option; it shows what to do in each state when you receive WILL/WONT/DO/DONT from the network, and when the user or process requests that an option be enabled or disabled. An implementor who uses the procedures given in that example need not worry about compliance with this RFC or with a large chunk of RFC 854.

In short, all implementors should be familiar with TELNET loops, and some implementors may wish to use the pre-written example here in

writing a new TELNET implementation.

NOTE: Reading This Document

A TELNET implementation is not compliant with this RFC if it fails to satisfy all rules marked MUST. It is compliant if it satisfies all rules marked MUST. If it is compliant, it is unconditionally compliant if it also satisfies all rules marked SHOULD and conditionally compliant otherwise. Rules marked MAY are optional.

Options are in almost all cases negotiated separately for each side of the connection. The option on one side is separate from the option on the other side. In this document, "the" option referred to by a DONT/WONT or DO/WILL is really two options, combined only for semantic convenience. Each sentence could be split into two, one with the words before the slash and one with the words after the slash.

An implementor should be able to determine whether or not an implementation complies with this RFC without reading any text marked DISCUSSION. An implementor should be able to implement option negotiation machinery compliant with both this RFC and RFC 854 using just the information in Section 7.

2. RFC 854 Option Negotiation Requirements

As specified by RFC 854: A TELNET implementation MUST obey a refusal to enable an option; i.e., if it receives a DONT/WONT in response to a WILL/DO, it MUST NOT enable the option.

DISCUSSION:

Where RFC 854 implies that the other side may reject a request to enable an option, it means that you must accept such a rejection.

It MUST therefore remember that it is negotiating a WILL/DO, and this negotiation state MUST be separate from the enabled state and from the disabled state. During the negotiation state, any effects of having the option enabled MUST NOT be used.

If it receives WONT/DONT and the option is enabled, it MUST respond DONT/WONT repectively and disable the option. It MUST NOT initiate a DO/WILL negotiation for an already enabled option or a DONT/WONT negotiation for a disabled option. It MUST NOT respond to receipt of such a negotiation. It MUST respond to receipt of a negotiation that does propose to change the status quo.

DISCUSSION:

Many existing implementations respond to rejection by confirming the rejection; i.e., if they send WILL and receive DONT, they send WONT. This has been construed as acceptable behavior under a certain (strained) interpretation of RFC 854. However, to allow this possibility severely complicates later rules; there seems to be no use for the wasted bandwidth and processing. Note that an implementation compliant with this RFC will simply ignore the extra WONT if the other side sends it.

The implementation MUST NOT automatically respond to the rejection of a request by submitting a new request. As a rule of thumb, new requests should be sent either at the beginning of a connection or in response to an external stimulus, i.e., input from the human user or from the process behind the server.

A TELNET implementation MUST refuse (DONT/WONT) a request to enable an option for which it does not comply with the appropriate protocol specification.

DISCUSSION:

This is not stated as strongly in RFC 854. However, any other action would be counterproductive. This rule appears in Requirements for Internet Hosts [6, Section 3.2.2]; it appears here for completeness.

3. Rule: Remember DONT/WONT requests

A TELNET implementation MUST remember starting a DONT/WONT negotiation.

DISCUSSION:

It is not clear from RFC 854 whether or not TELNET must remember beginning a DONT/WONT negotiation. There seem to be no reasons to remember starting a DONT/WONT negotiation: 1. The argument for remembering a DO/WILL negotiation (viz., the state of negotiating for enabling means different things for the data stream than the state of having the option enabled) does not apply. 2. There is no choice for the other side in responding to a DONT/WONT; the option is going to end up disabled. 3. If we simply disable the option immediately and forget negotiating, we will ignore the WONT/DONT response since the option is disabled.

Unfortunately, that conclusion is wrong. Consider the following TELNET conversation between two parties, "us" and "him". (The

reader of this RFC may want to sort the steps into chronological order for a different view.)

LOOP EXAMPLE 1

Both sides know that the option is on.

On his side:
1 He decides to disable. He sends DONT and disables the option.
2 He decides to reenable. He sends DO and remembers he is negotiating.
5 He receives WONT and gives up on negotiation.
6 He decides to try once again to reenable. He sends DO and remembers he is negotiating.
7 He receives WONT and gives up on negotiation.
For whatever reason, he decides to agree with future requests.
10 He receives WILL and agrees. He responds DO and enables the option.
11 He receives WONT and sighs. He responds DONT and disables the option.
(repeat 10 and then 11, forever)

On our side:
3 We receive DONT and sigh. We respond WONT and disable the option.
4 We receive DO but disagree. We respond WONT.
8 We receive DO and decide to agree. We respond WILL and enable the option.
9 We decide to disable. We send WONT and disable the option.
For whatever reason, we decide to agree with future requests.
12 We receive DO and agree. We send WILL and enable the option.
13 We receive DONT and sigh. We send WONT and disable the option.
(repeat 12 and then 13, forever)

Both sides have followed RFC 854; but we end in an option negotiation loop, as DONT DO DO and then DO DONT forever travel through the network one way, and WONT WONT followed by WILL WONT forever travel through the network the other way. The behavior in steps 1 and 9 is responsible for this loop. Hence this section's rule. In Section 6 below is discussion of whether separate states are needed for "negotiate for disable" and "negotiate for enable" or whether a single "negotiate" state suffices.

4. Rule: Prohibit new requests before completing old negotiation

A TELNET implementation MUST NOT initiate a new WILL/WONT/DO/DONT request about an option that is under negotiation, i.e., for which it has already made such a request and not yet received a response.

DISCUSSION:

It is unclear from RFC 854 whether or not a TELNET implementation may allow new requests about an option that is currently under negotiation; it certainly seems limiting to prohibit "option typeahead". Unfortunately, consider the following:

LOOP EXAMPLE 2

Suppose an option is disabled, and we decide in quick succession to enable it, disable it, and reenable it. We send WILL WONT WILL and at the end remember that we are negotiating. The other side agrees with DO DONT DO. We receive the first DO, enable the option, and forget we have negotiated. Now DONT DO are coming through the network and both sides have forgotten they are negotiating; consequently we loop.

(All possible TELNET loops eventually degenerate into the same form, where WILL WONT [or WONT WILL, or WILL WONT WILL WONT, etc.] go through the network while both sides think negotiation is over; the response is DO DONT and we loop forever. TELNET implementors are encouraged to implement any option that can detect such a loop and cut it off; e.g., a method of explicitly differentiating requests from acknowledgments would be sufficient. No such option exists as of February 1990.)

This particular case is of considerable practical importance: most combinations of existing user-server TELNET implementations do enter an infinite loop when asked quickly a few times to enable and then disable an option. This has taken on an even greater importance with the advent of LINEMODE [4], because LINEMODE is the first option that tends to generate such rapidly changing requests in the normal course of communication. It is clear that a new rule is needed.

One might try to prevent the several-alternating-requests problem by maintaining a more elaborate state than YES/NO/WANTwhatever, e.g., a state that records all outstanding requests. Dave Borman has proposed an apparently working scheme [2] that won’t blow up if both sides initiate several requests at once, and that seems to prevent option negotiation loops; complete analysis of his solution is somewhat difficult since it means that TELNET can no longer be a finite-state automaton. He has implemented his solution in the latest BSD telnet version [5]; as of May 1989, he does not intend to publish it for others to use [3].

Here the author decided to preserve TELNET’s finite-state property, for robustness and because the result can be easily

proven to work. Hence the above rule.

A more restrictive solution would be to buffer all data and do absolutely nothing until the response comes back. There is no apparent reason for this, though some existing TELNET implementations do so anyway at the beginning of a connection, when most options are negotiated.

5. How to reallow the request queue

DISCUSSION:

The above rule prevents queueing of more than one request through the network. However, it is possible to queue new requests within the TELNET implementation, so that "option typeahead" is effectively restored.

An obvious possibility is to maintain a list of requests that have been made but not yet sent, so that when one negotiation finishes, the next can be started immediately. So while negotiating for a WILL, TELNET could buffer the user's requests for WONT, then WILL again, then WONT, then WILL, then WONT, as far as desired.

This requires a dynamic and potentially unmanageable buffer. However, the restrictions upon possible requests guarantee that the list of requests must simply alternate between WONT and WILL. It is wasteful to enable an option and then disable it, just to enable it again; we might as well just enable it in the first place. The few possible exceptions to this rule do not outweigh the immense simplification afforded by remembering only the last few entries on the queue.

To be more precise, during a WILL negotiation, the only sensible queues are WONT and WONT WILL, and similarly during a WONT negotiation. In the interest of simplicity, the Q method does not allow the WONT WILL possibility.

We are now left with a queue consisting of either nothing or the opposite of the current negotiation. When we receive a reply to the negotiation, if the queue indicates that the option should be changed, we send the opposite request immediately and empty the queue. Note that this does not conflict with the RFC 854 rule about automatic regeneration of requests, as these new requests are simply the delayed effects of user or process commands.

An implementation SHOULD support the queue, where support is defined by the rules following.

If it does support the queue, and if an option is currently under negotiation, it MUST NOT handle a new request from the user or process to switch the state of that option by sending a new request through the network. Instead, it MUST remember internally that the new request was made.

If the user or process makes a second new request, for switching back again, while the original negotiation is still incomplete, the implementation SHOULD handle the request simply by forgetting the previous one. The third request SHOULD be treated like the first, etc. In any case, these further requests MUST NOT generate immediate requests through the network.

When the option negotiation completes, if the implementation is remembering a request internally, and that request is for the opposite state to the result of the completed negotiation, then the implementation MUST act as if that request had been made after the completion of the negotiation. It SHOULD thus immediately generate a new request through the network.

The implementation MAY provide a method by which support for the queue may be turned off and back on. In this case, it MUST default to having the support turned on. Furthermore, when support is turned off, if the implementation is remembering a new request for an outstanding negotiation, it SHOULD continue remembering and then deal with it at the close of the outstanding negotiation, as if support were still turned on through that point.

DISCUSSION:

It is intended (and it is the author’s belief) that this queue system restores the full functionality of TELNET. Dave Borman has provided some informal analysis of this issue [1]; the most important possible problem of note is that certain options which may require buffering could be slowed by the queue. The author believes that network delays caused by buffering are independent of the implementation method used, and that the Q Method does not cause any problems; this is borne out by examples.

6. Rule: Separate WANTNO and WANTYES

Implementations SHOULD separate any states of negotiating WILL/DO from any states of negotiating WONT/DONT.

DISCUSSION:

It is possible to maintain a working TELNET implementation if the NO/YES/WANTNO/WANTYES states are simplified to NO/YES/WANT.

However, in a hostile environment this is a bad idea, as it means that handling a DO/WILL response to a WONT/DONT cannot be done correctly. It does not greatly simplify code; and the simplicity gained is lost in the extra complexity needed to maintain the queue.

7. Example of Correct Implementation

To ease the task of writing TELNET implementations, the author presents here a precise example of the response that a compliant TELNET implementation could give in each possible situation. All TELNET implementations compliant with this RFC SHOULD follow the procedures shown here.

EXAMPLE STATE MACHINE
FOR THE Q METHOD OF IMPLEMENTING TELNET OPTION NEGOTIATION

There are two sides, we (us) and he (him). We keep four variables:

us: state of option on our side (NO/WANTNO/WANTYES/YES)
usq: a queue bit (EMPTY/OPPOSITE) if us is WANTNO or WANTYES
him: state of option on his side
himq: a queue bit if him is WANTNO or WANTYES

An option is enabled if and only if its state is YES. Note that us/usq and him/himq could be combined into two six-choice states.

"Error" below means that producing diagnostic information may be a good idea, though it isn't required.

Upon receipt of WILL, we choose based upon him and himq:

NO		If we agree that he should enable, him=YES, send DO; otherwise, send DONT.
YES		Ignore.
WANTNO	EMPTY	Error: DONT answered by WILL. him=NO.
	OPPOSITE	Error: DONT answered by WILL. him=YES*, himq=EMPTY.
WANTYES	EMPTY	him=YES.
	OPPOSITE	him=WANTNO, himq=EMPTY, send DONT.

* This behavior is debatable; DONT will never be answered by WILL over a reliable connection between TELNETs compliant with this RFC, so this was chosen (1) not to generate further messages, because if we know we're dealing with a noncompliant TELNET we shouldn't trust it to be sensible; (2) to empty the queue sensibly.

Upon receipt of WONT, we choose based upon him and himq:

NO		Ignore.
YES		him=NO, send DONT.
WANTNO	EMPTY	him=NO.
	OPPOSITE	him=WANTYES, himq=NONE, send DO.
WANTYES	EMPTY	him=NO.*
	OPPOSITE	him=NO, himq=NONE.**

* Here is the only spot a length-two queue could be useful; after a WILL negotiation was refused, a queue of WONT WILL would mean to request the option again. This seems of too little utility and too much potential waste; there is little chance that the other side will change its mind immediately.

** Here we don't have to generate another request because we've been "refused into" the correct state anyway.

If we decide to ask him to enable:

NO		him=WANTYES, send DO.
YES		Error: Already enabled.
WANTNO	EMPTY	If we are queueing requests, himq=OPPOSITE; otherwise, Error: Cannot initiate new request in the middle of negotiation.
	OPPOSITE	Error: Already queued an enable request.
WANTYES	EMPTY	Error: Already negotiating for enable.
	OPPOSITE	himq=EMPTY.

If we decide to ask him to disable:

NO		Error: Already disabled.
YES		him=WANTNO, send DONT.
WANTNO	EMPTY	Error: Already negotiating for disable.
	OPPOSITE	himq=EMPTY.
WANTYES	EMPTY	If we are queueing requests, himq=OPPOSITE; otherwise, Error: Cannot initiate new request in the middle of negotiation.
	OPPOSITE	Error: Already queued a disable request.

We handle the option on our side by the same procedures, with DO-WILL, DONT-WONT, him-us, himq-usq swapped.

8. References

[1] Borman, D., private communication, April 1989.

[2] Borman, D., private communication, May 1989.

[3] Borman, D., private communication, May 1989.

[4] Borman, D., Editor, "Telnet Linemode Option", RFC 1116, Cray Research, August 1989.

[5] Borman, D., BSD Telnet Source, November 1989.

[6] Braden, R., Editor, "Requirements for Internet Hosts -- Application and Support", RFC 1123, USC/Information Sciences Institute, October 1989.

[7] Postel, J., and J. Reynolds, "Telnet Protocol Specification", RFC 854, USC/Information Sciences Institute, May 1983.

9. Acknowledgments

Thanks to Dave Borman, dab@opus.cray.com, for his helpful comments.

Author's Address

Daniel J. Bernstein
5 Brewster Lane
Bellport, NY 11713

Phone: 516-286-1339

Email: brnstnd@acf10.nyu.edu