

## The Eifel Response Algorithm for TCP

### Status of This Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

### Copyright Notice

Copyright (C) The Internet Society (2005).

### Abstract

Based on an appropriate detection algorithm, the Eifel response algorithm provides a way for a TCP sender to respond to a detected spurious timeout. It adapts the retransmission timer to avoid further spurious timeouts and (depending on the detection algorithm) can avoid the often unnecessary go-back-N retransmits that would otherwise be sent. In addition, the Eifel response algorithm restores the congestion control state in such a way that packet bursts are avoided.

### 1. Introduction

The Eifel response algorithm relies on a detection algorithm such as the Eifel detection algorithm, defined in [RFC3522]. That document contains informative background and motivation context that may be useful for implementers of the Eifel response algorithm, but it is not necessary to read [RFC3522] in order to implement the Eifel response algorithm. Note that alternative response algorithms have been proposed [BA02] that could also rely on the Eifel detection algorithm, and alternative detection algorithms have been proposed [RFC3708], [SK04] that could work together with the Eifel response algorithm.

Based on an appropriate detection algorithm, the Eifel response algorithm provides a way for a TCP sender to respond to a detected spurious timeout. It adapts the retransmission timer to avoid

further spurious timeouts and (depending on the detection algorithm) can avoid the often unnecessary go-back-N retransmits that would otherwise be sent. In addition, the Eifel response algorithm restores the congestion control state in such a way that packet bursts are avoided.

Note: A previous version of the Eifel response algorithm also included a response to a detected spurious fast retransmit. However, as a consensus was not reached about how to adapt the duplicate acknowledgement threshold in that case, that part of the algorithm was removed for the time being.

### 1.1. Terminology

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in [RFC2119].

We refer to the first-time transmission of an octet as the 'original transmit'. A subsequent transmission of the same octet is referred to as a 'retransmit'. In most cases, this terminology can also be applied to data segments. However, when repacketization occurs, a segment can contain both first-time transmissions and retransmissions of octets. In that case, this terminology is only consistent when applied to octets. For the Eifel detection and response algorithms, this makes no difference, as they also operate correctly when repacketization occurs.

We use the term 'acceptable ACK' as defined in [RFC793]. That is an ACK that acknowledges previously unacknowledged data. We use the term 'bytes\_acked' to refer to the amount (in terms of octets) of previously unacknowledged data that is acknowledged by the most recently received acceptable ACK. We use the TCP sender state variables 'SND.UNA' and 'SND.NXT' as defined in [RFC793]. SND.UNA holds the segment sequence number of the oldest outstanding segment. SND.NXT holds the segment sequence number of the next segment the TCP sender will (re-)transmit. In addition, we define as 'SND.MAX' the segment sequence number of the next original transmit to be sent. The definition of SND.MAX is equivalent to the definition of 'snd\_max' in [WS95].

We use the TCP sender state variables 'cwnd' (congestion window), and 'ssthresh' (slow-start threshold), and the term 'FlightSize' as defined in [RFC2581]. FlightSize is the amount (in terms of octets) of outstanding data at a given point in time. We use the term 'Initial Window' (IW) as defined in [RFC3390]. The IW is the size of the sender's congestion window after the three-way handshake is completed. We use the TCP sender state variables 'SRTT' and

'RTTVAR', and the terms 'RTO' and 'G' as defined in [RFC2988]. G is the clock granularity of the retransmission timer. In addition, we assume that the TCP sender maintains the value of the latest round-trip time (RTT) measurement in the (local) variable 'RTT-SAMPLE'.

We use the TCP sender state variable 'T\_last', and the term 'tcpnow' as used in [RFC2861]. T\_last holds the system time when the TCP sender sent the last data segment, whereas tcpnow is the TCP sender's current system time.

## 2. Appropriate Detection Algorithms

If the Eifel response algorithm is implemented at the TCP sender, it **MUST** be implemented together with a detection algorithm that is specified in a standards track or experimental RFC.

Designers of detection algorithms who want their algorithms to work together with the Eifel response algorithm should reuse the variable "SpuriousRecovery" with the semantics and defined values specified in [RFC3522]. In addition, we define the constant LATE\_SPUR\_TO (set equal to -1) as another possible value of the variable SpuriousRecovery. Detection algorithms should set the value of SpuriousRecovery to LATE\_SPUR\_TO if the detection of a spurious retransmit is based on the ACK for the retransmit (as opposed to an ACK for an original transmit). For example, this applies to detection algorithms that are based on the DSACK option [RFC3708].

## 3. The Eifel Response Algorithm

The complete algorithm is specified in section 3.1. In sections 3.2 - 3.6, we discuss the different steps of the algorithm.

### 3.1. The Algorithm

Given that a TCP sender has enabled a detection algorithm that complies with the requirements set in Section 2, a TCP sender **MAY** use the Eifel response algorithm as defined in this subsection.

If the Eifel response algorithm is used, the following steps **MUST** be taken by the TCP sender, but only upon initiation of a timeout-based loss recovery. That is when the first timeout-based retransmit is sent. The algorithm **MUST NOT** be reinitiated after a timeout-based loss recovery has already been started but not completed. In particular, it may not be reinitiated upon subsequent timeouts for the same segment, or upon retransmitting segments other than the oldest outstanding segment.

- (0) Before the variables `cwnd` and `ssthresh` get updated when loss recovery is initiated, set a "pipe\_prev" variable as follows:  
    `pipe_prev <- max (FlightSize, ssthresh)`
- Set a "SRTT\_prev" variable and a "RTTVAR\_prev" variable as follows:  
    `SRTT_prev <- SRTT + (2 * G)`  
    `RTTVAR_prev <- RTTVAR`
- (DET) This is a placeholder for a detection algorithm that must be executed at this point, and that sets the variable `SpuriousRecovery` as outlined in Section 2. If [RFC3522] is used as the detection algorithm, steps (1) - (6) of that algorithm go here.
- (7) If `SpuriousRecovery` equals `SPUR_T0`, then  
    proceed to step (8);
- else if `SpuriousRecovery` equals `LATE_SPUR_T0`, then  
        proceed to step (9);
- else  
        proceed to step (DONE).
- (8) Resume the transmission with previously unsent data:  
Set  
    `SND.NXT <- SND.MAX`
- (9) Reverse the congestion control state:  
If the acceptable ACK has the ECN-Echo flag [RFC3168] set, then  
    proceed to step (DONE);
- else set  
        `cwnd <- FlightSize + min (bytes_acked, IW)`  
        `ssthresh <- pipe_prev`
- Proceed to step (DONE).
- (10) Interworking with Congestion Window Validation:  
If congestion window validation is implemented according to [RFC2861], then set  
    `T_last <- tcpnow`

(11) Adapt the conservativeness of the retransmission timer:

Upon the first RTT-SAMPLE taken from new data; i.e., the first RTT-SAMPLE that can be derived from an acceptable ACK for data that was previously unsent when the spurious timeout occurred,

if the retransmission timer is implemented according to [RFC2988], then set

```
SRTT    <- max (SRTT_prev, RTT-SAMPLE)
RTTVAR  <- max (RTTVAR_prev, RTT-SAMPLE/2)
RT0     <- SRTT + max (G, 4*RTTVAR)
```

Run the bounds check on the RT0 (rules (2.4) and (2.5) in [RFC2988]), and restart the retransmission timer;

else

appropriately adapt the conservativeness of the retransmission timer that is implemented.

(DONE) No further processing.

### 3.2. Storing the Current Congestion Control State (Step 0)

The TCP sender stores in `pipe_prev` what is considered a safe slow-start threshold (`ssthresh`) before loss recovery is initiated; i.e., before the loss indication is taken into account. This is either the current `FlightSize`, if the TCP sender is in congestion avoidance, or the current `ssthresh`, if the TCP sender is in slow-start. If the TCP sender later detects that it has entered loss recovery unnecessarily, then `pipe_prev` is used in step (9) to reverse the congestion control state. Thus, until the loss recovery phase is terminated, `pipe_prev` maintains a memory of the congestion control state of the time right before the loss recovery phase was initiated. A similar approach is proposed in [RFC2861], where this state is stored in `ssthresh` directly after a TCP sender has become idle or application limited.

There had been debates about whether the value of `pipe_prev` should be decayed over time; e.g., upon subsequent timeouts for the same outstanding segment. We do not require decaying `pipe_prev` for the Eifel response algorithm and do not believe that such a conservative approach should be in place. Instead, we follow the idea of revalidating the congestion window through slow-start, as suggested in [RFC2861]. That is, in step (9), the `cwnd` is reset to a value that avoids large packet bursts, and `ssthresh` is reset to the value of `pipe_prev`. Note that [RFC2581] and [RFC2861] also do not require

a decaying of `ssthresh` after it has been reset in response to a loss indication, or after a TCP sender has become idle or application limited.

### 3.3. Suppressing the Unnecessary go-back-N Retransmits (Step 8)

Without the use of the TCP timestamps option [RFC1323], the TCP sender suffers from the retransmission ambiguity problem [Zh86], [KP87]. Therefore, when the first acceptable ACK arrives after a spurious timeout, the TCP sender must assume that this ACK was sent in response to the retransmit when in fact it was sent in response to an original transmit. Furthermore, the TCP sender must further assume that all other segments that were outstanding at that point were lost.

Note: Except for certain cases where original ACKs were lost, the first acceptable ACK cannot carry a DSACK option [RFC2883].

Consequently, once the TCP sender's state has been updated after the first acceptable ACK has arrived, `SND.NXT` equals `SND.UNA`. This is what causes the often unnecessary go-back-N retransmits. From that point on every arriving acceptable ACK that was sent in response to an original transmit will advance `SND.NXT`. But as long as `SND.NXT` is smaller than the value that `SND.MAX` had when the timeout occurred, those ACKs will clock out retransmits, whether or not the corresponding original transmits were lost.

In fact, during this phase the TCP sender breaks 'packet conservation' [Jac88]. This is because the go-back-N retransmits are sent during slow-start. For each original transmit leaving the network, two retransmits are sent into the network as long as `SND.NXT` does not equal `SND.MAX` (see [LK00] for more detail).

Once a spurious timeout has been detected (upon receipt of an ACK for an original transmit), it is safe to let the TCP sender resume the transmission with previously unsent data. Thus, the Eifel response algorithm changes the TCP sender's state by setting `SND.NXT` to `SND.MAX`. Note that this step is only executed if the variable `SpuriousRecovery` equals `SPUR_T0`, which in turn requires a detection algorithm such as the Eifel detection algorithm [RFC3522] or the F-RT0 algorithm [SK04] that detects a spurious retransmit based upon receiving an ACK for an original transmit (as opposed to the ACK for the retransmit [RFC3708]).

### 3.4. Reversing the Congestion Control State (Step 9)

When a TCP sender enters loss recovery, it reduces `cwnd` and `ssthresh`. However, once the TCP sender detects that the loss recovery has been falsely triggered, this reduction proves unnecessary. We therefore believe that it is safe to revert to the previous congestion control state, following the approach of revalidating the congestion window as outlined below. This is unless the acceptable ACK signals congestion through the ECN-Echo flag [RFC3168]. In that case, the TCP sender MUST refrain from reversing congestion control state.

If the ECN-Echo flag is not set, `cwnd` is reset to the sum of the current `FlightSize` and the minimum of `bytes_acked` and `IW`. In some cases, this can mean that the first few acceptable ACKs that arrive will not clock out any data segments. Recall that `bytes_acked` is the number of bytes that have been acknowledged by the acceptable ACK. Note that the value of `cwnd` must not be changed any further for that ACK, and that the value of `FlightSize` at this point in time may be different from the value of `FlightSize` in step (0). The value of `IW` puts a limit on the size of the packet burst that the TCP sender may send into the network after the Eifel response algorithm has terminated. The value of `IW` is considered an acceptable burst size. It is the amount of data that a TCP sender may send into a yet "unprobed" network at the beginning of a connection.

Then `ssthresh` is reset to the value of `pipe_prev`. As a result, the TCP sender either immediately resumes probing the network for more bandwidth in congestion avoidance, or it slow-starts to what is considered a safe operating point for the congestion window.

### 3.5. Interworking with the CWV Algorithm (Step 10)

An implementation of the Congestion Window Validation (CWV) algorithm [RFC2861] could potentially misinterpret a delay spike that caused a spurious timeout as a phase where the TCP sender had been idle. Therefore, `T_last` is reset to prevent the triggering of the CWV algorithm in this case.

Note: The term 'idle' implies that the TCP sender has no data outstanding; i.e., all data sent has been acknowledged [Jac88]. According to this definition, a TCP sender is not idle while it is waiting for an acceptable ACK after a timeout. Unfortunately, the pseudo-code in [RFC2861] does not include a check for the condition "idle" (`SND.UNA == SND.MAX`). We therefore had to add step (10) to the Eifel response algorithm.

### 3.6. Adapting the Retransmission Timer (Step 11)

There is currently only one retransmission timer standardized for TCP [RFC2988]. We therefore only address that timer explicitly. Future standards that might define alternatives to [RFC2988] should propose similar measures to adapt the conservativeness of the retransmission timer.

A spurious timeout often results from a delay spike, which is a sudden increase of the RTT that usually cannot be predicted. After a delay spike, the RTT may have changed permanently; e.g., due to a path change, or because the available bandwidth on a bandwidth-dominated path has decreased. This may often occur with wide-area wireless access links. In this case, the RTT estimators (SRTT and RTTVAR) should be reinitialized from the first RTT-SAMPLE taken from new data according to rule (2.2) of [RFC2988]. That is, from the first RTT-SAMPLE that can be derived from an acceptable ACK for data that was previously unsent when the spurious timeout occurred.

However, a delay spike may only indicate a transient phase, after which the RTT returns to its previous range of values, or even to smaller values. Also, a spurious timeout may occur because the TCP sender's RTT estimators were only inaccurate enough that the retransmission timer expires "a tad too early". We believe that two times the clock granularity of the retransmission timer ( $2 * G$ ) is a reasonable upper bound on "a tad too early". Thus, when the new RT0 is calculated in step (11), we ensure that it is at least ( $2 * G$ ) greater (see also step (0)) than the RT0 was before the spurious timeout occurred.

Note that other TCP sender processing will usually take place between steps (10) and (11). During this phase (i.e., before step (11) has been reached), the RT0 is managed according to the rules of [RFC2988]. We believe that this is sufficiently conservative for the following reasons. First, the retransmission timer is restarted upon the acceptable ACK that was used to detect the spurious timeout. As a result, the delay spike is already implicitly factored in for segments outstanding at that time. This is discussed in more detail in [EL04], where this effect is called the "RT0 offset". Furthermore, if timestamps are enabled, a new and valid RTT-SAMPLE can be derived from that acceptable ACK. This RTT-SAMPLE must be relatively large, as it includes the delay spike that caused the spurious timeout. Consequently, the RTT estimators will be updated rather conservatively. Without timestamps the RT0 will stay conservatively backed-off due to Karn's algorithm [RFC2988] until the first RTT-SAMPLE can be derived from an acceptable ACK for data that was previously unsent when the spurious timeout occurred.



For the new RTT to become effective, the retransmission timer has to be restarted. This is consistent with [RFC2988], which recommends restarting the retransmission timer with the arrival of an acceptable ACK.

#### 4. Advanced Loss Recovery is Crucial for the Eifel Response Algorithm

We have studied environments where spurious timeouts and multiple losses from the same flight of packets often coincide [GL02], [GL03]. In such a case, the oldest outstanding segment arrives at the TCP receiver, but one or more packets from the remaining outstanding flight are lost. In those environments, end-to-end performance suffers if the Eifel response algorithm is operated without an advanced loss recovery scheme such as a SACK-based scheme [RFC3517] or NewReno [RFC3782]. The reason is TCP-Reno's aggressiveness after a spurious timeout. Even though TCP-Reno breaks 'packet conservation' (see Section 3.3) when blindly retransmitting all outstanding segments, it usually recovers all packets lost from that flight within a single round-trip time. On the contrary, the more conservative TCP-Reno-with-Eifel is often forced into another timeout. Thus, we recommend that the Eifel response algorithm always be operated in combination with [RFC3517] or [RFC3782]. Additional robustness is achieved with the Limited Transmit and Early Retransmit algorithms [RFC3042], [AAAB04].

Note: The SACK-based scheme we used for our simulations in [GL02] and [GL03] is different from the SACK-based scheme that later got standardized [RFC3517]. The key difference is that [RFC3517] is more robust to multiple losses from the same flight. It is less conservative in declaring that a packet has left the network, and is therefore less dependent on timeouts to recover genuine packet losses.

If the NewReno algorithm [RFC3782] is used in combination with the Eifel response algorithm, step (1) of the NewReno algorithm SHOULD be modified as follows, but only if SpuriousRecovery equals SPUR\_T0:

(1) Three duplicate ACKs:

When the third duplicate ACK is received and the sender is not already in the Fast Recovery procedure, go to step 1A.

That is, the entire step 1B of the NewReno algorithm is obsolete because step (8) of the Eifel response algorithm avoids the case where three duplicate ACKs result from unnecessary go-back-N retransmits after a timeout. Step (8) of the Eifel response algorithm avoids such unnecessary go-back-N retransmits in the first place. However, recall that step (8) is only executed if the variable SpuriousRecovery equals SPUR\_T0, which in turn requires a

detection algorithm, such as the Eifel detection algorithm [RFC3522] or the F-RTT algorithm [SK04], that detects a spurious retransmit based upon receiving an ACK for an original transmit (as opposed to the ACK for the retransmit [RFC3708]).

## 5. Security Considerations

There is a risk that a detection algorithm is fooled by spoofed ACKs that make genuine retransmits appear to the TCP sender as spurious retransmits. When such a detection algorithm is run together with the Eifel response algorithm, this could effectively disable congestion control at the TCP sender. Should this become a concern, the Eifel response algorithm SHOULD only be run together with detection algorithms that are known to be safe against such "ACK spoofing attacks".

For example, the safe variant of the Eifel detection algorithm [RFC3522], is a reliable method to protect against this risk.

## 6. Acknowledgements

Many thanks to Keith Sklower, Randy Katz, Michael Meyer, Stephan Baucke, Sally Floyd, Vern Paxson, Mark Allman, Ethan Blanton, Pasi Sarolahti, Alexey Kuznetsov, and Yogesh Swami for many discussions that contributed to this work.

## 7. References

### 7.1. Normative References

- [RFC2581] Allman, M., Paxson, V., and W. Stevens, "TCP Congestion Control", RFC 2581, April 1999.
- [RFC3390] Allman, M., Floyd, S., and C. Partridge, "Increasing TCP's Initial Window", RFC 3390, October 2002.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3782] Floyd, S., Henderson, T., and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 3782, April 2004.
- [RFC2861] Handley, M., Padhye, J., and S. Floyd, "TCP Congestion Window Validation", RFC 2861, June 2000.
- [RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, April 2003.

- [RFC2988] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.
- [RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

## 7.2. Informative References

- [RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.
- [AAAB04] Allman, M., Avrachenkov, K., Ayesta, U., and J. Blanton, "Early Retransmit for TCP and SCTP", Work in Progress, July 2004.
- [BA02] Blanton, E. and M. Allman, "On Making TCP More Robust to Packet Reordering", ACM Computer Communication Review, Vol. 32, No. 1, January 2002.
- [RFC3708] Blanton, E. and M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions", RFC 3708, February 2004.
- [RFC3517] Blanton, E., Allman, M., Fall, K., and L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP", RFC 3517, April 2003.
- [EL04] Ekstrom, H. and R. Ludwig, "The Peak-Hopper: A New End-to-End Retransmission Timer for Reliable Unicast Transport", In Proceedings of IEEE INFOCOM 04, March 2004.
- [RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgment (SACK) Option for TCP", RFC 2883, July 2000.
- [GL02] Gurtov, A. and R. Ludwig, "Evaluating the Eifel Algorithm for TCP in a GPRS Network", In Proceedings of the European Wireless Conference, February 2002.
- [GL03] Gurtov, A. and R. Ludwig, "Responding to Spurious Timeouts in TCP", In Proceedings of IEEE INFOCOM 03, April 2003.

- [Jac88] Jacobson, V., Congestion Avoidance and Control, In Proceedings of ACM SIGCOMM 88.
- [RFC1323] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.
- [KP87] Karn, P. and C. Partridge, Improving Round-Trip Time Estimates in Reliable Transport Protocols, In Proceedings of ACM SIGCOMM 87.
- [LK00] Ludwig, R. and R. H. Katz, The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions, ACM Computer Communication Review, Vol. 30, No. 1, January 2000.
- [SK04] Sarolahti, P. and M. Kojo, F-RTT: An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and SCTP, Work in Progress, November 2004.
- [WS95] Wright, G. R. and W. R. Stevens, TCP/IP Illustrated, Volume 2 (The Implementation), Addison Wesley, January 1995.
- [Zh86] Zhang, L., Why TCP Timers Don't Work Well, In Proceedings of ACM SIGCOMM 88.

#### Authors' Addresses

Reiner Ludwig  
Ericsson Research (EDD)  
Ericsson Allee 1  
52134 Herzogenrath, Germany

EMail: Reiner.Ludwig@ericsson.com

Andrei Gurtov  
Helsinki Institute for Information Technology (HIIT)  
P.O. Box 9800, FIN-02015  
HUT, Finland

EMail: andrei.gurtov@cs.helsinki.fi  
Homepage: <http://www.cs.helsinki.fi/u/gurtov>

## Full Copyright Statement

Copyright (C) The Internet Society (2005).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the IETF's procedures with respect to rights in IETF Documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at [ietf-ipr@ietf.org](mailto:ietf-ipr@ietf.org).

## Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.