Network Working Group
Internet-Draft
Updates: 6126bis (if approved)
Intended status: Standards Track
Expires: April 11, 2019

A. Decimo
IRIF, University of Paris-Diderot
D. Schinazi
Apple Inc.
J. Chroboczek
IRIF, University of Paris-Diderot
October 8, 2018

Babel Routing Protocol over Datagram Transport Layer Security
draft-ietf-babel-dtls-01

Abstract

The Babel Routing Protocol does not contain any means to authenticate neighbours or protect messages sent between them. This documents describes a mechanism to ensure these properties, using Datagram Transport Layer Security (DTLS). This document updates RFC 6126bis.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 11, 2019.

Table of Contents

1. Introduction

The Babel Routing Protocol [RFC6126bis] does not contain any means to authenticate neighbours or protect messages sent between them. Because of this, an attacker is able to send maliciously crafted Babel messages which could lead a network to route traffic to an attacker or to an under-resourced target causing denial of service. This documents describes a mechanism to prevent such attacks, using Datagram Transport Layer Security (DTLS) [RFC6347].

1.1. Specification of Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Applicability

The current two main mechanisms for securing Babel are Babel over DTLS (as described in this document) and Babel Cryptographic Authentication [BabelHMAC]. The latter has the advantages of being simpler and not requiring a dependency on DTLS, therefore implementers are encouraged to consider it in preference to the

mechanism defined in this document whenever both are applicable to a given deployment. Both mechanisms ensure integrity of messages and prevent message replay.

However, DTLS offers several features that are not provided by Babel Cryptographic Authentication, therefore Babel over DTLS is applicable in cases where those features are needed. Examples of such features include:

- Asymmetric keys. DTLS allows authentication via asymmetric keys, which allows a finer granularity of trust per-peer, and allows for revocation.

- Confidentiality of data. DTLS encrypts payloads, preventing an on-link attacker from observing the routing table.

2. Operation of the Protocol

Babel over DTLS requires changes to how Babel is operated, for two reasons. Firstly, because DTLS introduces the concepts of client and server, while Babel is a peer-to-peer protocol. Secondly, DTLS can only protect unicast, while Babel TLVs can be sent over both unicast and multicast.

2.1. DTLS Connection Initiation

All Babel over DTLS nodes MUST act as DTLS servers on the "babel-dtls" port (UDP port TBD), and MUST listen for multicast traffic on the unencrypted "babel" port (UDP port 6696). When a Babel node discovers a new neighbor (generally by receiving an unencrypted multicast Babel packet), it compares the neighbour's IPv6 link-local address with its own, using network byte ordering. If a node's address is lower than the recently discovered neighbor's address, it acts as a client and connects to the neighbor. In other words, the node with the lowest address is the DTLS client for this pairwise relationship. As an example, fe80::1:2 is considered lower than fe80::2:1. The node acting as DTLS client initiates its DTLS connection from an ephemeral UDP port. Nodes SHOULD ensure that new client DTLS connections use different ephemeral ports from recently used connections to allow servers to differentiate between the new and old DTLS connections. When a node receives a new DTLS connection, it MUST verify the source IP address, and reject the connection if the address is not an IPv6 link-local address.

2.2. Protocol Encoding

Babel over DTLS sends all unicast Babel packets encrypted by DTLS. The entire Babel packet, from the Magic byte at the start of the Babel header to the last byte of the Babel packet trailer, is sent protected by DTLS.

2.3. Transmission

When sending packets, Babel over DTLS nodes MUST NOT send any TLVs over the unprotected "babel" port, with the exception of Hello TLVs without the Unicast flag set. Babel over DTLS nodes MUST NOT send any unprotected unicast packet. Unless some out-of-band neighbor discovery mechanism is available, nodes SHOULD periodically send unprotected multicast Hellos to ensure discovery of new neighbours. In order to maintain bidirectional reachability, nodes can either rely on unprotected multicast Hellos, or also send protected unicast Hellos.

Since Babel over DTLS only protects unicast packets, implementors may implement Babel over DTLS by modifying an unprotected implementation of Babel, and replacing any TLV sent over multicast with a separate TLV sent over unicast for each neighbour.

2.4. Reception

Babel over DTLS nodes can receive Babel packets either protected over a DTLS connection, or unprotected directly over the "babel" port. To ensure the security properties of this mechanism, unprotected packets are treated differently. Nodes MUST silently ignore any unprotected packet sent over unicast. When parsing an unprotected packet, a node MUST silently ignore all TLVs that are not of type Hello. Nodes MUST also silently ignore any unprotected Hello with the Unicast flag set. Note that receiving an unprotected packet can still be used to discover new neighbors, even when all TLVs in that packet are silently ignored.

2.5. Neighbour table entry

It is RECOMMENDED for nodes to associate the state of their DTLS connection with their neighbour table. When a neighbour entry is flushed from the neighbour table (Appendix A of [RFC6126bis]), its associated DTLS state SHOULD be discarded. The node MAY send a DTLS close_notify alert to the neighbour.

3. Interface Maximum Transmission Unit Issues

Compared to unprotected Babel, DTLS adds header, authentication tag and possibly block-size padding overhead to every packet. This reduces the size of the Babel payload that can be carried. Nodes SHOULD compute the overhead of DTLS depending on the ciphers in use, and SHOULD NOT send Babel packets larger than the interface maximum transmission unit (MTU) minus the overhead of lower layers (IP, UDP and DTLS). This helps reduce the likelihood of lower-layer fragmentation which would negatively impact performance and reliability. Nodes MUST NOT send Babel packets larger than the attached interface's MTU adjusted for known lower-layer headers (at least UDP and IP) or 512 octets, whichever is larger, but not exceeding 2^16 - 1 adjusted for lower-layer headers. Every Babel speaker MUST be able to receive packets that are as large as any attached interface's MTU adjusted for UDP and IP headers or 512 octets, whichever is larger. Note that this requirement on reception does not take into account the overhead of DTLS because the peer may not have the ability to compute the overhead of DTLS and the packet may be fragmented by lower layers. Babel packets MUST NOT be sent in IPv6 Jumbograms.

4. IANA Considerations

If this document is approved, IANA is requested to register a UDP port number, called "babel-dtls", for use by Babel over DTLS.

5. Security Considerations

The interaction between two Babel peers requires Datagram Transport Layer Security (DTLS) with a cipher suite offering confidentiality protection. The guidance given in [RFC7525] MUST be followed to avoid attacks on DTLS. The DTLS client SHOULD use the TLS Certificate Status Request extension (Section 8 of [RFC6066]).

A malicious client might attempt to perform a high number of DTLS handshakes with a server. As the clients are not uniquely identified by the protocol and can be obfuscated with IPv4 address sharing and with IPv6 temporary addresses, a server needs to mitigate the impact of such an attack. Such mitigation might involve rate limiting handshakes from a given subnet or more advanced denial of service avoidance techniques beyond the scope of this document.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6126bis] Chroboczek, J. and D. Schinazi, "The Babel Routing Protocol", Internet Draft draft-ietf-babel-rfc6126bis-05, May 2018.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

6.2. Informative References

[BabelHMAC] Do, C., Kolodziejak, W., and J. Chroboczek, "Babel Cryptographic Authentication", Internet Draft draft-ietf-babel-hmac-00, August 2018.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/rfc7525>.

[RFC7918] Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", RFC 7918, DOI 10.17487/RFC7918, August 2016, <https://www.rfc-editor.org/info/rfc7918>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC8094] Reddy, T., Wing, D., and P. Patil, "DNS over Datagram Transport Layer Security (DTLS)", RFC 8094, DOI 10.17487/RFC8094, February 2017, <https://www.rfc-editor.org/info/rfc8094>.

Appendix A. Performance Considerations

To reduce the number of octets taken by the DTLS handshake, especially the size of the certificate in the ServerHello (which can be several kilobytes), Babel peers can use raw public keys [RFC7250] or the Cached Information Extension [RFC7924]. The Cached Information Extension avoids transmitting the server's certificate and certificate chain if the client has cached that information from a previous TLS handshake. TLS False Start [RFC7918] can reduce round trips by allowing the TLS second flight of messages (ChangeCipherSpec) to also contain the (encrypted) Babel packet.

These performance considerations were inspired from the ones for DNS over DTLS [RFC8094].

Authors' Addresses

Antonin Decimo
IRIF, University of Paris-Diderot
Paris
France

Email: antonin.decimo@gmail.com

David Schinazi
Apple Inc.
One Apple Park Way
Cupertino, California 95014
USA

Email: dschinazi@apple.com

Juliusz Chroboczek
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: jch@irif.fr

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 1, 2021

A. Decimo
IRIF, University of Paris-Diderot
D. Schinazi
Google LLC
J. Chroboczek
IRIF, University of Paris-Diderot

June 30, 2020

Babel Routing Protocol over Datagram Transport Layer Security
draft-ietf-babel-dtls-10

Abstract

The Babel Routing Protocol does not contain any means to authenticate neighbours or provide integrity or confidentiality for messages sent between them. This document specifies a mechanism to ensure these properties, using Datagram Transport Layer Security (DTLS).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 1, 2021.

Table of Contents

1. Introduction

The Babel Routing Protocol [RFC6126bis] does not contain any means to authenticate neighbours or protect messages sent between them. Because of this, an attacker is able to send maliciously crafted Babel messages which could lead a network to route traffic to an attacker or to an under-resourced target causing denial of service. This document specifies a mechanism to prevent such attacks, using Datagram Transport Layer Security (DTLS) [RFC6347].

1.1. Specification of Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Applicability

The protocol described in this document protects Babel packets with DTLS. As such, it inherits the features offered by DTLS, notably authentication, integrity, optional replay protection, confidentiality and asymmetric keying. It is therefore expected to be applicable in a wide range of environments.

There exists another mechanism for securing Babel, namely Babel HMAC authentication [BABEL-HMAC]. HMAC only offers basic features, namely authentication, integrity and replay protection with a small number of symmetric keys. A comparison of Babel security mechanisms and their applicability can be found in [RFC6126bis].

Note that Babel over DTLS provides a single authentication domain, meaning that all nodes that have the right credentials can convey any and all routing information.

DTLS supports several mechanisms by which nodes can identify themselves and prove possession of secrets tied to these identities. This document does not prescribe which of these mechanisms to use; details of identity management are left to deployment profiles of Babel over DTLS.

2. Operation of the Protocol

Babel over DTLS requires some changes to how Babel operates. First, DTLS is a client-server protocol, while Babel is a peer-to-peer protocol. Second, DTLS can only protect unicast communication, while Babel packets can be sent to both unicast and multicast destinations.

2.1. DTLS Connection Initiation

Babel over DTLS operates on a different port than unencrypted Babel. All Babel over DTLS nodes MUST act as DTLS servers on a given UDP port, and MUST listen for unencrypted Babel traffic on another UDP port, which MUST be distinct from the first one. The default port for Babel over DTLS is registered with IANA as the "babel-dtls" port (UDP port TBD, see Section 4), and the port exchanging unencrypted Babel traffic is registered as the "babel" port (UDP port 6696, see Section 5 of [RFC6126bis]).

When a Babel node discovers a new neighbour (generally by receiving an unencrypted multicast Babel packet), it compares the neighbour's IP address with its own, using network byte ordering. If a node's address is lower than the recently discovered neighbour's address, it acts as a client and connects to the neighbour. In other words, the node with the lowest address is the DTLS client for this pairwise

relationship. As an example, fe80::1:2 is considered lower than fe80::2:1.

The node acting as DTLS client initiates its DTLS connection from an ephemeral UDP port. Nodes SHOULD ensure that new client DTLS connections use different ephemeral ports from recently used connections to allow servers to differentiate between the new and old DTLS connections. Alternatively, nodes could use DTLS connection identifiers [DTLS-CID] as a higher-entropy mechanism to distinguish between connections.

When a node receives a new DTLS connection, it MUST verify that the source IP address is either an IPv6 link-local address or an IPv4 address belonging to the local network; if it is neither, it MUST reject the connection. Nodes use mutual authentication (authenticating both client and server); clients MUST authenticate servers and servers MUST authenticate clients. Implementations MUST support authenticating peers against a local store of credentials. If either node fails to authenticate its peer against its local policy, it MUST abort the DTLS handshake. The guidance given in [BCP195] MUST be followed to avoid attacks on DTLS. Additionally, nodes MUST only negotiate DTLS version 1.2 or higher. Nodes MUST use DTLS replay protection to prevent attackers from replaying stale information. Nodes SHOULD drop packets that have been reordered by more than two IHU (I Heard You) intervals, to avoid letting attackers make stale information last longer. If a node receives a new DTLS connection from a neighbour to whom it already has a connection, the node MUST NOT discard the older connection until it has completed the handshake of the new one and validated the identity of the peer.

2.2. Protocol Encoding

Babel over DTLS sends all unicast Babel packets protected by DTLS. The entire Babel packet, from the Magic byte at the start of the Babel header to the last byte of the Babel packet trailer, is sent protected by DTLS.

2.3. Transmission

When sending packets, Babel over DTLS nodes MUST NOT send any TLVs over the unprotected "babel" port, with the exception of Hello TLVs without the Unicast flag set. Babel over DTLS nodes MUST NOT send any unprotected unicast packets. This ensures the confidentiality of the information sent in Babel packets (e.g., the network topology) by only sending it encrypted by DTLS. Unless some out-of-band neighbour discovery mechanism is available, nodes SHOULD periodically send unprotected multicast Hellos to ensure discovery of new neighbours. In order to maintain bidirectional reachability, nodes can either

rely entirely on unprotected multicast Hellos, or send protected unicast Hellos in addition to the multicast Hellos.

Since Babel over DTLS only protects unicast packets, implementors may implement Babel over DTLS by modifying an implementation of Babel without DTLS support, and replacing any TLV previously sent over multicast with a separate TLV sent over unicast for each neighbour. TLVs previously sent over multicast can be replaced with the same contents over unicast, with the exception of Hellos as described above. Some implementations could also change the contents of IHU TLVs when converting to unicast in order to remove redundant information.

2.4. Reception

Babel over DTLS nodes can receive Babel packets either protected over a DTLS connection, or unprotected directly over the "babel" port. To ensure the security properties of this mechanism, unprotected packets are treated differently. Nodes MUST silently ignore any unprotected packet sent over unicast. When parsing an unprotected packet, a node MUST silently ignore all TLVs that are not of type Hello. Nodes MUST also silently ignore any unprotected Hello with the Unicast flag set. Note that receiving an unprotected packet can still be used to discover new neighbours, even when all TLVs in that packet are silently ignored.

2.5. Neighbour table entry

It is RECOMMENDED for nodes to associate the state of their DTLS connection with their neighbour table. When a neighbour entry is flushed from the neighbour table (Appendix A of [RFC6126bis]), its associated DTLS state SHOULD be discarded. The node SHOULD send a DTLS close_notify alert to the neighbour if it believes the link is still viable.

2.6. Simultaneous operation of both Babel over DTLS and unprotected Babel on a Node

Implementations MAY implement both Babel over DTLS and unprotected Babel. Additionally, a node MAY simultaneously run both Babel over DTLS and unprotected Babel. However, a node running both MUST ensure that it runs them on separate interfaces, as the security properties of Babel over DTLS rely on not accepting unprotected Babel packets (other than multicast Hellos). An implementation MAY offer configuration options to allow unprotected Babel on some interfaces but not others; this effectively gives nodes on that interface the same access as authenticated nodes, and SHOULD NOT be done unless

that interface has a mechanism to authenticate nodes at a lower layer (e.g., IPsec).

2.7. Simultaneous operation of both Babel over DTLS and unprotected Babel on a Network

If Babel over DTLS and unprotected Babel are both operated on the same network, the Babel over DTLS implementation will receive unprotected multicast Hellos and attempt to initiate a DTLS connection. These connection attempts can be sent to nodes that only run unprotected Babel, who will not respond. Babel over DTLS implementations SHOULD therefore rate-limit their DTLS connection attempts to avoid causing undue load on the network.

3. Interface Maximum Transmission Unit Issues

Compared to unprotected Babel, DTLS adds header, authentication tag and possibly block-size padding overhead to every packet. This reduces the size of the Babel payload that can be carried. This document does not relax the packet size requirements in Section 4 of [RFC6126bis], but recommends that DTLS overhead be taken into account when computing maximum packet size.

More precisely, nodes SHOULD compute the overhead of DTLS depending on the ciphersuites in use, and SHOULD NOT send Babel packets larger than the interface maximum transmission unit (MTU) minus the overhead of IP, UDP and DTLS. Nodes MUST NOT send Babel packets larger than the attached interface's MTU adjusted for known lower-layer headers (at least UDP and IP) or 512 octets, whichever is larger, but not exceeding 2^16 - 1 adjusted for lower-layer headers. Every Babel speaker MUST be able to receive packets that are as large as any attached interface's MTU adjusted for UDP and IP headers or 512 octets, whichever is larger. Note that this requirement on reception does not take into account the overhead of DTLS because the peer may not have the ability to compute the overhead of DTLS and the packet may be fragmented by lower layers.

Note that distinct DTLS connections can use different ciphers, which can have different amounts of per-packet overhead. Therefore, the MTU to one neighbour can be different from the MTU to another neighbour on the same link.

4. IANA Considerations

If this document is approved, IANA is requested to register a UDP port number, called "babel-dtls", for use by Babel over DTLS. Details of the request to IANA are as follows:

- Assignee: IESG, iesg@ietf.org
- Contact Person: IETF Chair, chair@ietf.org
- Transport Protocols: UDP only
- Service Code: None
- Service Name: babel-dtls
- Desired Port Number: 6699
- Description: Babel Routing Protocol over DTLS
- Reference: This document
- Defined TXT Keys: None

5. Security Considerations

A malicious client might attempt to perform a high number of DTLS handshakes with a server. As the clients are not uniquely identified by the protocol until the handshake completes and can be obfuscated with IPv6 temporary addresses, a server needs to mitigate the impact of such an attack. Note that attackers might attempt to keep in-progress handshakes open for as long as possible by using variants on the attack commonly known as Slowloris [SLOWLORIS]. Mitigating these attacks might involve rate limiting handshakes from a given subnet or more advanced denial of service avoidance techniques beyond the scope of this document.

Babel over DTLS allows sending multicast Hellos unprotected; attackers can therefore tamper with them. For example, an attacker could send erroneous values for the Seqno and Interval fields, causing bidirectional reachability detection to fail. While implementations MAY use multicast Hellos for link quality estimation, they SHOULD also emit protected unicast Hellos to prevent this class of denial-of-service attack.

While DTLS provides protection against an attacker that replays valid packets, DTLS is not able to detect when an active on-path attacker intercepts valid packets and resends them at a later time. This attack could be used to make a node believe it has bidirectional reachability to a neighbour even though that neighbour has disconnected from the network. To prevent this attack, nodes MUST discard the DTLS state associated with a neighbour after a finite time of not receiving valid DTLS packets. This can be implemented by, for example, discarding a neighbour's DTLS state when its

associated IHU timer fires. Note that relying solely on the receipt of Hellos is not sufficient as multicast Hellos are sent unprotected. Additionally, an attacker could save some packets and replay them later in hopes of propagating stale routing information at a later time. This can be mitigated by discarding received packets that have been reordered by more than two IHU intervals.

6. References

6.1. Normative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/bcp195>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6126bis] Chroboczek, J. and D. Schinazi, "The Babel Routing Protocol", Internet Draft draft-ietf-babel-rfc6126bis-17, February 2020.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

6.2. Informative References

[BABEL-HMAC] Do, C., Kolodziejak, W., and J. Chroboczek, "Babel Cryptographic Authentication", Internet Draft draft-ietf-babel-hmac-10, August 2019.

[DTLS-CID] Rescorla, E., Tschofenig, H., Fossati, T., and T. Gondrom, "Connection Identifiers for DTLS 1.2", Internet Draft draft-ietf-tls-dtls-connection-id-07, October 2019.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

[RFC7918] Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", RFC 7918, DOI 10.17487/RFC7918, August 2016, <https://www.rfc-editor.org/info/rfc7918>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC8094] Reddy, T., Wing, D., and P. Patil, "DNS over Datagram Transport Layer Security (DTLS)", RFC 8094, DOI 10.17487/RFC8094, February 2017, <https://www.rfc-editor.org/info/rfc8094>.

[SLOWLORIS] Hansen, R., "Welcome to Slowloris...", June 2009, <https://web.archive.org/web/20150315054838/http://ha.ckers.org/slowloris/>.

Appendix A. Performance Considerations

To reduce the number of octets taken by the DTLS handshake, especially the size of the certificate in the ServerHello (which can be several kilobytes), Babel peers can use raw public keys [RFC7250] or the Cached Information Extension [RFC7924]. The Cached Information Extension avoids transmitting the server's certificate and certificate chain if the client has cached that information from a previous TLS handshake. TLS False Start [RFC7918] can reduce round trips by allowing the TLS second flight of messages (ChangeCipherSpec) to also contain the (encrypted) Babel packet.

Appendix B. Acknowledgments

The authors would like to thank Roman Danyliw, Donald Eastlake, Thomas Fossati, Benjamin Kaduk, Gabriel Kerneis, Mirja Kuehlewind, Antoni Przygienda, Henning Rogge, Dan Romascanu, Barbara Stark, Markus Stenberg, Dave Taht, Martin Thomson, Sean Turner and Martin Vigoureux for their input and contributions. The performance considerations in this document were inspired from the ones for DNS over DTLS [RFC8094].

Authors' Addresses

Antonin Decimo
IRIF, University of Paris-Diderot
Paris
France

Email: antonin.decimo@gmail.com

David Schinazi
Google LLC
1600 Amphitheatre Parkway
Mountain View, California 94043
USA

Email: dschinazi.ietf@gmail.com

Juliusz Chroboczek
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: jch@irif.fr

Babel routing protocol
Internet-Draft
Intended status: Informational
Expires: April 25, 2019

B. Stark
AT&T

October 22, 2018

Babel Information Model
draft-ietf-babel-information-model-04

Abstract

This Babel Information Model can be used to create data models under various data modeling regimes (e.g., YANG). It allows a Babel implementation (via a management protocol such as NETCONF) to report on its current state and may allow some limited configuration of protocol constants.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 25, 2019.

Table of Contents

1. Introduction

Babel is a loop-avoiding distance-vector routing protocol defined in [I-D.ietf-babel-rfc6126bis]. [I-D.ietf-babel-hmac] defines a security mechanism that allows Babel messages to be cryptographically authenticated, and [I-D.ietf-babel-dtls] defines a security mechanism that allows Babel messages to be encrypted. This document describes an information model for Babel (including implementations using one of these security mechanisms) that can be used to create management protocol data models (such as a NETCONF [RFC6241] YANG [RFC7950] data model).

Due to the simplicity of the Babel protocol, most of the information model is focused on reporting Babel protocol operational state, and very little of that is considered mandatory to implement (contingent on a management protocol with Babel support being implemented). Some parameters may be configurable. However, it is up to the Babel implementation whether to allow any of these to be configured within its implementation. Where the implementation does not allow configuration of these parameters, it may still choose to expose them as read-only.

The Information Model is presented using a hierarchical structure. This does not preclude a data model based on this Information Model from using a referential or other structure.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119] and updated by [RFC8174].

1.2. Notation

This document uses a programming language-like notation to define the properties of the objects of the information model. An optional property is enclosed by square brackets, [ ], and a list property is indicated by two numbers in angle brackets, <m..n>, where m indicates the minimal number of list elements, and n indicates the maximum number of list elements. The symbol * for n means there are no defined limits on the number of list elements. Each parameter and object includes an indication of "ro" or "rw". "ro" means the parameter or object is read-only. "rw" means it is read-write. For an object, read-write means instances of the object can be created or deleted. If an implementation is allowed to choose to implement a "rw" parameter as read-only, this is noted in the parameter description.

The object definitions use base types that are defined as follows:

binary A binary string (sequence of octets).

boolean A type representing a boolean value.

counter A non-negative integer that monotonically increases. Counters may have discontinuities and they are not expected to persist across restarts.

credentials An opaque type representing credentials needed by a cryptographic mechanism to secure communication. Data models must expand this opaque type as needed and required by the security protocols utilized.

datetime A type representing a date and time using the Gregorian calendar. The datetime format MUST conform to RFC 3339 [RFC3339].

int A type representing signed or unsigned integer numbers. This information model does not define a precision nor

does it make a distinction between signed and unsigned number ranges.

ip-address A type representing an IP address. This type supports both IPv4 and IPv6 addresses.

string A type representing a human-readable string consisting of a (possibly restricted) subset of Unicode and ISO/IEC 10646 [ISO.10646] characters.

uri A type representing a Uniform Resource Identifier as defined in STD 66 [RFC3986].

2. Overview

The Information Model is hierarchically structured as follows:

```
information object
   includes implementation version, router id, this node seqno,
     enable flag parameters, supported security mechanisms
   constants object (exactly one per information object)
      includes UDP port and optional multicast group
        parameters
   interfaces object
      includes interface reference, Hello seqno and intervals,
        update interval, link type, metric computation parameters
      neighbors object
         includes neighbor IP address, Hello history, cost
           parameters
      security object (per interface)
         includes enable flag, self credentials (credential
           object), trusted credentials (credential object)
   security object (common to all interfaces)
      includes enable flag, self credentials (credential
        object), trusted credentials (credential object)
   routes object
      includes route prefix, source router, reference to
        advertising neighbor, metric, sequence number, whether
        route is feasible, whether route is selected
```

Most parameters are read-only. Following is a list of the parameters that are not required to be read-only:

- enable/disable Babel
- Constant: UDP port
- Constant: IPv6 multicast group

- Interface: Link type
- Interface: External cost (must be configurable if implemented, but implementation is optional)
- Interface: enable/disable Babel on this interface
- Interface: enable/disable message log
- Security: enable/disable this security mechanism
- Security: self credentials
- Security: trusted credentials
- Security: enable/disable security log

Note that this overview is intended simply to be informative and is not normative. If there is any discrepancy between this overview and the detailed information model definitions in subsequent sections, the error is in this overview.

3. The Information Model

3.1. Definition of babel-information-obj

```
  object {
       string               ro babel-implementation-version;
       boolean              rw babel-enable;
       binary               ro babel-self-router-id;
       string               ro babel-supported-link-types<1..*>;
      [int                  ro babel-self-seqno;]
       string               ro babel-metric-comp-algorithms<1..*>;
       string               ro babel-security-supported<0..*>;
       babel-constants-obj  ro babel-constants;
       babel-interfaces-obj ro babel-interfaces<0..*>;
       babel-routes-obj     ro babel-routes<0..*>;
       babel-security-obj   ro babel-security<0..*>;
   } babel-information-obj;
```

babel-implementation-version: The name and version of this implementation of the Babel protocol.

babel-enable: When written, it configures whether the protocol shoud be enabled (true) or disabled (false). A read from the running or intended datastore indicates the configured administrative value of whether the protocol is enabled (true) or not (false). A read from the operational datastore indicates whether the protocol is

actually running (true) or not (i.e., it indicates the operational state of the protocol). A data model that does not replicate parameters for running and operational datastores can implement this as two separate parameters. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-self-router-id: The router-id used by this instance of the Babel protocol to identify itself. [I-D.ietf-babel-rfc6126bis] describes this as an arbitrary string of 8 octets.

babel-supported-link-types: Lists the set of link types supported by this instance of Babel. Valid enumeration values are defined in the Babel Link Types registry (see Section 7).

babel-self-seqno: The current sequence number included in route updates for routes originated by this node.

babel-metric-comp-algorithms: List of supported cost computation algorithms. Possible values include "k-out-of-j", and "ETX".

babel-security-supported: List of supported security mechanisms. As Babel security mechanisms are defined, they will need to indicate what enumeration value is to be used to represent them in this parameter.

babel-constants: A babel-constants-obj object.

babel-interfaces: A set of babel-interface-obj objects.

babel-security: A babel-security-obj object that applies to all interfaces. If this object is implemented, it allows a security mechanism to be enabled or disabled in a manner that applies to all Babel messages on all interfaces.

babel-routes: A set of babel-route-obj objects. Contains the routes known to this node.

3.2. Definition of babel-constants-obj

```
object {
     int          rw babel-udp-port;
    [ip-address   rw babel-mcast-group;]
 } babel-constants-obj;
```

babel-udp-port: UDP port for sending and listening for Babel messages. Default is 6696. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-mcast-group: Multicast group for sending and listening to multicast announcements on IPv6. Default is ff02:0:0:0:0:0:1:6. An implementation MAY choose to expose this parameter as read-only ("ro").

3.3. Definition of babel-interfaces-obj

```
  object {
       string               ro babel-interface-reference;
      [boolean              rw babel-interface-enable;]
       int                  rw babel-link-type;
       string               ro babel-interface-metric-algorithm;
      [int                  ro babel-mcast-hello-seqno;]
      [int                  ro babel-mcast-hello-interval;]
      [int                  ro babel-update-interval;]
      [boolean              rw babel-message-log-enable;]
      [babel-log-obj        ro babel-message-log<0..*>;]
       babel-neighbors-obj  ro babel-neighbors<0..*>;
      [babel-security-obj   ro babel-interface-security<0..*>;]
   } babel-interfaces-obj;
```

babel-interface-reference: Reference to an interface object as defined by the data model (e.g., YANG [RFC7950], BBF [TR-181]). Data model is assumed to allow for referencing of interface objects which may be at any layer (physical, Ethernet MAC, IP, tunneled IP, etc.). referencing syntax will be specific to the data model. If there is no set of interface objects available, this should be a string that indicates the interface name used by the underlying operating system.

babel-interface-enable: When written, it configures whether the protocol should be enabled (true) or disabled (false) on this interface. A read from the running or intended datastore indicates the configured administrative value of whether the protocol is enabled (true) or not (false). A read from the operational datastore indicates whether the protocol is actually running (true) or not (i.e., it indicates the operational state of the protocol). A data model that does not replicate parameters for running and operational datastores can implement this as two separate parameters. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-link-type: Indicates the type of link. Valid enumeration values are identified in Babel Link Types registry. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-interface-metric-algorithm: Indicates the metric computation algorithm used on this interface. The value MUST be one of those listed in the babel-information-obj babel-metric-comp-algorithms parameter.

babel-mcast-hello-seqno: The current sequence number in use for multicast hellos sent on this interface.

babel-mcast-hello-interval: The current interval in use for multicast hellos sent on this interface.

babel-update-interval: The current interval in use for all updates (multicast and unicast) sent on this interface.

babel-message-log-enable: When written, it configures whether logging should be enabled (true) or disabled (false). A read from the running or intended datastore indicates the configured administrative value of whether logging is enabled (true) or not (false). A read from the operational datastore indicates whether logging is actually running (true) or not (i.e., it indicates the operational state). A data model that does not replicate parameters for running and operational datastores can implement this as two separate parameters. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-message-log: Log entries that have timestamp of a received Babel message and the entire received Babel message (including Ethernet frame and IP headers, if possible). An implementation must restrict the size of this log, but how and what size is implementation-specific. If this log is implemented, a mechanism to clear it SHOULD be provided.

babel-neighbors: A set of babel-neighbors-obj objects.

babel-interface-security: A babel-security-obj object that applies to this interface. If implemented, this allows security to be enabled only on specific interfaces or allows different security mechanisms to be enabled on different interfaces.

3.4. Definition of babel-neighbors-obj

```
  object {
       ip-address        ro babel-neighbor-address;
      [binary            ro babel-hello-mcast-history;]
      [binary            ro babel-hello-ucast-history;]
       int               ro babel-txcost;
       int               ro babel-exp-mcast-hello-seqno;
       int               ro babel-exp-ucast-hello-seqno;
      [int               ro babel-ucast-hello-seqno;]
      [int               ro babel-ucast-hello-interval;]
      [int               ro babel-rxcost]
      [int               ro babel-cost]
   } babel-neighbors-obj;
```

babel-neighbor-address: IPv4 or IPv6 address the neighbor sends messages from

babel-hello-mcast-history: The multicast Hello history of whether or not the multicast Hello messages prior to babel-exp-mcast-hello-seqno were received. A binary sequence where the most recently received Hello is expressed as a "1" placed in the left-most bit, with prior bits shifted right (and "0" bits placed between prior Hello bits and most recent Hello for any not-received Hellos). This value should be displayed using hex digits ([0-9a-fA-F]). See [I-D.ietf-babel-rfc6126bis], section A.1.

babel-hello-ucast-history: The unicast Hello history of whether or not the unicast Hello messages prior to babel-exp-ucast-hello-seqno were received. A binary sequence where the most recently received Hello is expressed as a "1" placed in the left-most bit, with prior bits shifted right (and "0" bits placed between prior Hello bits and most recent Hello for any not-received Hellos). This value should be displayed using hex digits ([0-9a-fA-F]). See [I-D.ietf-babel-rfc6126bis], section A.1.

babel-txcost: Transmission cost value from the last IHU packet received from this neighbor, or maximum value (infinity) to indicate the IHU hold timer for this neighbor has expired. See [I-D.ietf-babel-rfc6126bis], section 3.4.2.

babel-exp-mcast-hello-seqno: Expected multicast Hello sequence number of next Hello to be received from this neighbor. If multicast Hello messages are not expected, or processing of multicast messages is not enabled, this MUST be 0.

babel-exp-ucast-hello-seqno: Expected unicast Hello sequence number of next Hello to be received from this neighbor. If unicast Hello messages are not expected, or processing of unicast messages is not enabled, this MUST be 0.

babel-ucast-hello-seqno: The current sequence number in use for unicast hellos sent to this neighbor.

babel-ucast-hello-interval: The current interval in use for unicast hellos sent to this neighbor.

babel-rxcost: Reception cost calculated for this neighbor. This value is usually derived from the Hello history, which may be combined with other data, such as statistics maintained by the link layer. The rxcost is sent to a neighbor in each IHU. See [I-D.ietf-babel-rfc6126bis], section 3.4.3.

babel-cost: Link cost is computed from the values maintained in the neighbor table: the statistics kept in the neighbor table about the reception of Hellos, and the txcost computed from received IHU packets.

3.5. Definition of babel-security-obj

```
  object {
       string                ro babel-security-mechanism
       boolean               rw babel-security-enable;
       babel-credential-obj  ro babel-security-self-cred<0..*>;
       babel-credential-obj  ro babel-security-trust<0..*>;
      [boolean               rw babel-credvalid-log-enable;]
      [babel-log-obj         ro babel-credvalid-log<0..*>;]
   } babel-security-obj;
```

babel-security-mechanism: The name of the security mechanism this object instance is about. The value MUST be the same as one of the enumerations listed in the babel-security-supported parameter.

babel-security-enable: When written, it configures whether this security mechanism should be enabled (true) or disabled (false). A read from the running or intended datastore indicates the configured administrative value of whether this security mechanism is enabled (true) or not (false). A read from the operational datastore indicates whether this security mechanism is actually running (true) or not (i.e., it indicates the operational state). A data model that does not replicate parameters for running and operational datastores can implement this as two separate parameters. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-security-self-cred: Credentials this router presents to participate in the enabled security mechanism. Any private key component of a credential MUST NOT be readable. Adding and deleting credentials MAY be allowed.

babel-security-trust: A set of babel-credential-obj objects that identify the credentials of routers whose Babel messages may be trusted or of a certificate authority (CA) whose signing of a router's credentials implies the router credentials can be trusted, in the context of this security mechanism. How a security mechanism interacts with this list is determined by the mechanism. A security algorithm may do additional validation of credentials, such as checking validity dates or revocation lists, so presence in this list may not be sufficient to determine trust. Adding and deleting credentials MAY be allowed.

babel-credvalid-log-enable: When written, it configures whether logging should be enabled (true) or disabled (false). A read from the running or intended datastore indicates the configured administrative value of whether logging is enabled (true) or not (false). A read from the operational datastore indicates whether logging is actually running (true) or not (i.e., it indicates the operational state). A data model that does not replicate parameters for running and operational datastores can implement this as two separate parameters. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-credvalid-log: Log entries that have the timestamp a message containing credentials used for peer authentication (e.g., DTLS Server Hello) was received on a Babel port, and the entire received message (including Ethernet frame and IP headers, if possible). An implementation must restrict the size of this log, but how and what size is implementation-specific. If this log is implemented, a mechanism to clear it SHOULD be provided.

3.6. Definition of babel-routes-obj

```
  object {
       ip-address           ro babel-route-prefix;
       int                  ro babel-route-prefix-length;
       binary               ro babel-route-router-id;
       string               ro babel-route-neighbor;
      [int                  ro babel-route-received-metric;]
      [int                  ro babel-route-calculated-metric;]
       int                  ro babel-route-seqno;
       ip-address           ro babel-route-next-hop;
       boolean              ro babel-route-feasible;
       boolean              ro babel-route-selected;
   } babel-routes-obj;
```

babel-route-prefix: Prefix (expressed in IP address format) for which this route is advertised.

babel-route-prefix-length: Length of the prefix for which this route is advertised babel-route-router-id: router-id of the source router for which this route is advertised.

babel-route-neighbor: Reference to the babel-neighbors entry for the neighbor that advertised this route.

babel-route-received-metric: The metric with which this route was advertised by the neighbor, or maximum value (infinity) to indicate the route was recently retracted and is temporarily unreachable (see Section 3.5.5 of [I-D.ietf-babel-rfc6126bis]). This metric will be 0 (zero) if the route was not received from a neighbor but was generated through other means. Either babel-route-calculated-metric or babel-route-received-metric MUST be provided.

babel-route-calculated-metric: A calculated metric for this route. How the metric is calculated is implementation-specific. Maximum value (infinity) indicates the route was recently retracted and is temporarily unreachable (see Section 3.5.5 of [I-D.ietf-babel-rfc6126bis]). Either babel-route-calculated-metric or babel-route-received-metric MUST be provided.

babel-route-seqno: The sequence number with which this route was advertised.

babel-route-next-hop: The next-hop address of this route. This will be empty if this route has no next-hop address.

babel-route-feasible: A boolean flag indicating whether this route is feasible, as defined in Section 3.5.1 of [I-D.ietf-babel-rfc6126bis]).

babel-route-selected: A boolean flag indicating whether this route is selected (i.e., whether it is currently being used for forwarding and is being advertised).

4. Common Objects

4.1. Definition of babel-credential-obj

```
  object {
       credentials          ro babel-cred;
   } babel-credential-obj;
```

babel-cred: A credential, such as an X.509 certificate, a public key, etc. used for signing and/or encrypting Babel messages.

4.2. Definition of babel-log-obj

```
        object {
             datetime           ro babel-log-time;
             string             ro babel-log-entry;
         } babel-log-obj;
```

babel-log-time: The date and time (according to the device internal clock setting, which may be a time relative to boot time, acquired from NTP, configured by the user, etc.) when this log entry was created.

babel-log-entry: The logged message, as a string of utf-8 encoded hex characters.

5. Extending the Information Model

Implementations MAY extend this information model with other parameters or objects. For example, an implementation MAY choose to expose Babel route filtering rules by adding a route filtering object with parameters appropriate to how route filtering is done in that implementation. The precise means used to extend the information model would be specific to the data model the implementation uses to expose this information.

6. Security Considerations

This document defines a set of information model objects and parameters that may be exposed to be visible from other devices, and some of which may be configured. Securing access to and ensuring the integrity of this data is in scope of and the responsibility of any data model derived from this information model. Specifically, any YANG [RFC7950] data model is expected to define security exposure of the various parameters, and a [TR-181] data model will be secured by the mechanisms defined for the management protocol used to transport it.

This information model defines objects that can allow credentials (for this device, for trusted devices, and for trusted certificate authorities) to be added and deleted. Public keys and shared secrets may be exposed through this model. This model requires that private keys never be exposed. The Babel security mechanisms that make use of these credentials (e.g., [I-D.ietf-babel-dtls], [I-D.ietf-babel-hmac]) are expected to define what credentials can be used with those mechanisms.

7. IANA Considerations

This document defines a Babel Link Type registry for the values of the babel-link-type and babel-supported-link-types parameters to be listed under the Babel Routing Protocol registry.

Valid Babel Link Type names are normatively defined as

- MUST be at least 1 character and no more than 20 characters long
- MUST contain only US-ASCII [RFC0020] letters 'A' - 'Z' and 'a' - 'z', digits '0' - '9', and hyphens ('-', ASCII 0x2D or decimal 45)
- MUST contain at least one letter ('A' - 'Z' or 'a' - 'z')
- MUST NOT begin or end with a hyphen
- hyphens MUST NOT be adjacent to other hyphens

The rules for Link Type names, excepting the limit of 20 characters maximum, are also expressed below (as a non-normative convenience) using ABNF [RFC5234].

```
SRVNAME = *(1*DIGIT [HYPHEN]) ALPHA *([HYPHEN] ALNUM)
ALNUM   = ALPHA / DIGIT     ; A-Z, a-z, 0-9
HYPHEN  = %x2D              ; "-"
ALPHA   = %x41-5A / %x61-7A ; A-Z / a-z [RFC5234]
DIGIT   = %x30-39           ; 0-9       [RFC5234]
```

The allocation policy of this registry is Specification Required [RFC8126].

The initial values in the "Babel Link Type" registry are:

Name	Used for Links Defined By	Reference
ethernet	[IEEE-802.3-2018]	(this document)
other	to be used when no link type information available	(this document)
tunnel	to be used for a tunneled interface over unknown physical link	(this document)
wireless	[IEEE-802.11-2016]	(this document)
exp-*	Reserved for Experimental Use	(this document)

8. Acknowledgements

Juliusz Chroboczek, Toke Hoeiland-Joergensen, David Schinazi, Mahesh Jethanandani, Acee Lindem, and Carsten Bormann have been very helpful in refining this information model.

The language in the Notation section was mostly taken from [RFC8193].

9. References

9.1. Normative References

[I-D.ietf-babel-rfc6126bis]
Chroboczek, J. and D. Schinazi, "The Babel Routing Protocol", draft-ietf-babel-rfc6126bis-05 (work in progress), May 2018.

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

9.2. Informative References

[I-D.ietf-babel-dtls]
Decimo, A., Schinazi, D., and J. Chroboczek, "Babel Routing Protocol over Datagram Transport Layer Security", draft-ietf-babel-dtls-01 (work in progress), October 2018.

[I-D.ietf-babel-hmac]
Do, C., Kolodziejak, W., and J. Chroboczek, "Babel Cryptographic Authentification", draft-ietf-babel-hmac-00 (work in progress), August 2018.

[IEEE-802.11-2016]
"IEEE Standard 802.11-2016 - IEEE Standard for Information Technology - Telecommunications and information exchange between systems Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.".

[IEEE-802.3-2018]
"IEEE Standard 802.3-2018 - IEEE Approved Draft Standard for Ethernet.".

[ISO.10646]
International Organization for Standardization, "Information Technology - Universal Multiple-Octet Coded Character Set (UCS)", ISO Standard 10646:2014, 2014.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/info/rfc6241>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8193] Burbridge, T., Eardley, P., Bagnulo, M., and J. Schoenwaelder, "Information Model for Large-Scale Measurement Platforms (LMAPs)", RFC 8193, DOI 10.17487/RFC8193, August 2017, <https://www.rfc-editor.org/info/rfc8193>.

[TR-181] Broadband Forum, "Device Data Model", <http://cwmp-data-models.broadband-forum.org/>.

Appendix A. Open Issues

1. I want to get rid of the security log, because all Babel messages (which should be defined as all messages to/from the udp-port) are be logged by message-log. I don't like message log as it is. I think if logging is enabled it should just write to a text file. This will mean there also needs to be a means of downloading/reading the log file.

2. Consider the following statistics: under interface object: sent multicast Hello, sent updates, received Babel messages; under neighbor object: sent unicast Hello, sent updates, sent IHU, received Hello, received updates, received IHUs. Would also need to enable/disable stats and clear stats.

3. Security section needs furter review

4. Commands to add and delete credentials, and parameters that allow credential to be identified without allowing access to private credential info

5. Check description of enable parameters to make sure ok for YANG and TR-181. Closed by updating description to be useful for YANG and TR-181, using language consistent with YANG descriptions.

6. Distinguish signed and unsigned integers?

7. Review new IANA Considerations section. Should ABNF be normative?

Closed Issues:

1. Datatype of the router-id: Closed by introducing binary datatype and using that for router-id

2. babel-neighbor-address as IPv6-only: Closed by leaving as is (IPv4 and IPv6)

3. babel-implementation-version includes the name of the implementation: Closed by adding "name" to description

4. Delete external-cost?: Closed by deleting.

5. Would it be useful to define some parameters for reporting statistics or logs? [2 logs are now included. If others are needed they need to be proposed. See Open Issues for additional thoughts on logs and statistics.]

6. Closed by defining base64 type and using it for all router IDs: "babel-self-router-id: Should this be an opaque 64-bit value instead of int?"

7. Closed as "No": Do we need a registry for the supported security mechanisms? [Given the current limited set, and unlikelihood of massive expansion, I don't think so. But we can if someone wants it.]

8. This draft must be reviewed against draft-ietf-babel-rfc6126bis. [I feel like this has been adequately done, but I could be wrong.]

9. babel-interfaces-obj: Juliusz:"This needs further discussion, I fear some of these are implementation details." [In the absence of discussion, the current model stands. Note that all but link-type and the neighbors sub-object are optional. If an implementation does not have any of the optional elements then it simply doesn't have them and that's fine.]

10. Would it be useful to define some parameters specifically for security anomalies? [The 2 logs should be useful in identifying security anomalies. If more is needed, someone needs to propose.]

11. I created a basic security model. It's useful for single (or no) active security mechanism (e.g., just HMAC, just DTLS, or neither); but not multiple active (both HMAC and DTLS -- which is not the same as HMAC of DTLS and would just mean that HMAC would be used on all unencrypted messages -- but right now the model doesn't allow for configuring HMAC of unencrypted messages for routers without DTLS, while DTLS is used if possible). OK? [No-one said otherwise.]

12. babel-external-cost may need more work. [if no comment, it will be left as is]

13. babel-hello-[mu]cast-history: the Hello history is formated as 16 bits, per A.1 of 6126bis. Is that a too implementation specific? [We also now have an optional-to-implement log of received messages, and I made these optional. So maybe this is ok?]

14. rxcost, txcost, cost: is it ok to model as integers, since 6126bis 2.1 says costs and metrics need not be integers. [I have them as integers unless someone insists on something else.]

15. For the security log, should it also log whether the credentials were considered ok? [Right now it doesn't and I think that's ok because if you log Hellos it was ok and if you don't it wasn't.]

16. Should Babel link types have an IANA registry? [Agreed to do this at IETF 102.]

Appendix B. Change Log

Individual Drafts:

v00 2016-07-07 EBD: Initial individual draft version

v01 2017-03-13: Addressed comments received in 2016-07-15 email from J. Chroboczek

Working group drafts:

v00 2017-07-03: Addressed points noted with "oops" in https://www.ietf.org/proceedings/98/slides/slides-98-babel-babel-information-model-00.pdf

v01 2018-01-02: Removed item from issue list that was agreed (in Prague) not to be an issue. Added description of data types under Notation section, and used these in all data types. Added babel-security and babel-trust.

v02 2018-04-05:

* changed babel-version description to babel-implementation-version

* replace optional babel-interface-seqno with optional babel-mcast-hello-seqno and babel-ucast-hello-seqno

* replace optional babel-interface-hello-interval with optional babel-mcast-hello-interval and babel-ucast-hello-interval

* remove babel-request-trigger-ack

* remove "babel-router-id: router-id of the neighbor"; note that parameter had previously been removed but description had accidentally not been removed

* added an optional "babel-cost" field to babel-neighbors object, since the spec does not define how exactly the cost is computed from rxcost/txcost

* deleted babel-source-garbage-collection-time

* change babel-lossy-link to babel-link-type and make this an enumeration; added at top level babel-supported-link-types so which are supported by this implementation can be reported

* changes to babel-security-obj to allow self credentials to be one or more instances of a credential object. Allowed trusted credentials to include CA credentials; made some parameter name changes

* updated references and Introduction

* added Overview section

* deleted babel-sources-obj

* added feasible Boolean to routes

* added section to briefly describe extending the information model.

* deleted babel-route-neighbor

* tried to make definition of babel-interface-reference clearer

* added security and message logs

v03 2018-05-31:

* added reference to RFC 8174 (update to RFC 2119 on key words)

* applied edits to Introduction text per Juliusz email of 2018-04-06

* Deleted sentence in definition of "int" data type that said it was also used for enumerations. Changed all enumerations to strings. The only enumerations were for link types, which are now "ethernet", "wireless", "tunnel", and "other".

* deleted [ip-address babel-mcast-group-ipv4;]

* babel-external-cost description changed

* babel-security-self-cred: Added "any private key component of a credential MUST NOT be readable;"

* hello-history parameters put recent Hello in most significant bit and length of parameter is not constrained.

* babel-hello-seqno in neighbors-obj changed to babel-exp-mcast-hello-seqno and babel-exp-ucast-hello-seqno

* added babel-route-neighbor back again. It was mistakenly deleted

* changed babel-route-metric and babel-route-announced-metric to babel-route-received-metric and babel-route-calculated-metric

* changed model of security object to put list of supported mechanisms at top level and separate security object per mechanism. This caused some other changes to the security object

v04 2018-10-15:

* changed babel-mcast-group-ipv6 to babel-mcast-group

* link type parameters changed to point to newly defined registry

* babel-ucast-hello-interval moved to neighbor object

* babel-ucast-hello-seqno moved to neighbor object

* babel-neighbor-ihu-interval deleted

* in log descriptions, included statement that there SHOULD be ability to clear logs

* added IANA registry for link types

* added "ro" and "rw" to tables for read-write and read-only

* added metric computation parameter to interface

Author's Address

Barbara Stark
AT&T
Atlanta, GA
US

Email: barbara.stark@att.com

Babel routing protocol
Internet-Draft
Intended status: Informational
Expires: February 15, 2021

B. Stark
AT&T
M. Jethanandani
VMware

August 14, 2020

Babel Information Model
draft-ietf-babel-information-model-11

Abstract

This Babel Information Model provides structured data elements for a Babel implementation reporting its current state and may allow limited configuration of some such data elements. This information model can be used as a basis for creating data models under various data modeling regimes.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on February 15, 2021.

Table of Contents

1. Introduction

Babel is a loop-avoiding distance-vector routing protocol defined in [I-D.ietf-babel-rfc6126bis]. [I-D.ietf-babel-hmac] defines a security mechanism that allows Babel packets to be cryptographically authenticated, and [I-D.ietf-babel-dtls] defines a security mechanism that allows Babel packets to be encrypted. This document describes an information model for Babel (including implementations using one or both of these security mechanisms) that can be used to create management protocol data models (such as a NETCONF [RFC6241] YANG [RFC7950] data model.

Due to the simplicity of the Babel protocol, most of the information model is focused on reporting Babel protocol operational state, and very little of that is considered mandatory to implement for an implementation claiming compliance with this information model. Some parameters may be configurable. However, it is up to the Babel implementation whether to allow any of these to be configured within its implementation. Where the implementation does not allow

configuration of these parameters, it MAY still choose to expose them as read-only.

The Information Model is presented using a hierarchical structure. This does not preclude a data model based on this Information Model from using a referential or other structure.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119] and updated by [RFC8174].

1.2. Notation

This document uses a programming language-like notation to define the properties of the objects of the information model. An optional property is enclosed by square brackets, [ ], and a list property is indicated by two numbers in angle brackets, <m..n>, where m indicates the minimal number of list elements, and n indicates the maximum number of list elements. The symbol * for n means there are no defined limits on the number of list elements. Each parameter and object includes an indication of "ro" or "rw". "ro" means the parameter or object is read-only. "rw" means it is read-write. For an object, read-write means instances of the object can be created or deleted. If an implementation is allowed to choose to implement a "rw" parameter as read-only, this is noted in the parameter description.

The object definitions use base types that are defined as follows:

binary — A binary string (sequence of octets).

boolean — A type representing a Boolean (true or false) value.

counter — A non-negative integer that monotonically increases. Counters may have discontinuities and they are not expected to persist across restarts.

datetime — A type representing a date and time using the Gregorian calendar. The datetime format MUST conform to RFC 3339 [RFC3339].

ip-address — A type representing an IP address. This type supports both IPv4 and IPv6 addresses.

operation A type representing a remote procedure call or other action that can be used to manipulate data elements or system behaviors.

reference A type representing a reference to another information or data model element or to some other device resource.

string A type representing a human-readable string consisting of a (possibly restricted) subset of Unicode and ISO/IEC 10646 [ISO.10646] characters.

uint A type representing an unsigned integer number. This information model does not define a precision.

2. Overview

The Information Model is hierarchically structured as follows:

```
+-- babel-information
   +-- babel-implementation-version
   +-- babel-enable
   +-- router-id
   +-- self-seqno
   +-- babel-metric-comp-algorithms
   +-- babel-security-supported
   +-- babel-mac-algorithms
   +-- babel-dtls-cert-types
   +-- babel-stats-enable
   +-- babel-stats-reset
   +-- babel-constants
   |  +-- babel-udp-port
   |  +-- babel-mcast-group
   +-- babel-interfaces
   |  +-- babel-interface-reference
   |  +-- babel-interface-enable
   |  +-- babel-interface-metric-algorithm
   |  +-- babel-interface-split-horizon
   |  +-- babel-mcast-hello-seqno
   |  +-- babel-mcast-hello-interval
   |  +-- babel-update-interval
   |  +-- babel-mac-enable
   |  +-- babel-if-mac-key-sets
   |  +-- babel-mac-verify
   |  +-- babel-dtls-enable
   |  +-- babel-if-dtls-cert-sets
   |  +-- babel-dtls-cached-info
   |  +-- babel-dtls-cert-prefer
   |  +-- babel-packet-log-enable
```

```
      |  +-- babel-packet-log
      |  +-- babel-if-stats
      |  |  +-- babel-sent-mcast-hello
      |  |  +-- babel-sent-mcast-update
      |  |  +-- babel-sent-ucast-hello
      |  |  +-- babel-sent-ucast-update
      |  |  +-- babel-sent-IHU
      |  |  +-- babel-received-packets
      |  +-- babel-neighbors
      |     +-- babel-neighbor-address
      |     +-- babel-hello-mcast-history
      |     +-- babel-hello-ucast-history
      |     +-- babel-txcost
      |     +-- babel-exp-mcast-hello-seqno
      |     +-- babel-exp-ucast-hello-seqno
      |     +-- babel-ucast-hello-seqno
      |     +-- babel-ucast-hello-interval
      |     +-- babel-rxcost
      |     +-- babel-cost
      +-- babel-routes
      |  +-- babel-route-prefix
      |  +-- babel-route-prefix-length
      |  +-- babel-route-router-id
      |  +-- babel-route-neighbor
      |  +-- babel-route-received-metric
      |  +-- babel-route-calculated-metric
      |  +-- babel-route-seqno
      |  +-- babel-route-next-hop
      |  +-- babel-route-feasible
      |  +-- babel-route-selected
      +-- babel-mac-key-sets
      |  +-- babel-mac-default-apply
      |  +-- babel-mac-keys
      |     +-- babel-mac-key-name
      |     +-- babel-mac-key-use-sign
      |     +-- babel-mac-key-use-verify
      |     +-- babel-mac-key-value
      |     +-- babel-mac-key-algorithm
      |     +-- babel-mac-key-test
      +-- babel-dtls-cert-sets
         +-- babel-dtls-default-apply
         +-- babel-dtls-certs
            +-- babel-cert-name
            +-- babel-cert-value
            +-- babel-cert-type
            +-- babel-cert-private-key
            +-- babel-cert-test
```

Most parameters are read-only. Following is a descriptive list of the parameters that are not required to be read-only:

- enable/disable Babel
- create/delete Babel MAC Key sets
- create/delete Babel DTLS Certificate sets
- enable/disable statistics collection
- Constant: UDP port
- Constant: IPv6 multicast group
- Interface: Metric algorithm
- Interface: Split horizon
- Interface: enable/disable Babel on this interface
- Interface: sets of MAC keys
- Interface: MAC algorithm
- Interface: verify received MAC packets
- Interface: set of DTLS certificates
- Interface: use cached info extensions
- Interface: preferred order of certificate types
- Interface: enable/disable packet log
- MAC-keys: create/delete entries
- MAC-keys: key used to sign packets
- MAC-keys: key used to verify packets
- DTLS-certs: create/delete entries

The following parameters are required to return no value when read:

- MAC key values
- DTLS certificate values

Note that this overview is intended simply to be informative and is not normative. If there is any discrepancy between this overview and the detailed information model definitions in subsequent sections, the error is in this overview.

3. The Information Model

3.1. Definition of babel-information-obj

```
object {
     string                    ro babel-implementation-version;
     boolean                   rw babel-enable;
     binary                    ro babel-self-router-id;
    [uint                      ro babel-self-seqno;]
     string                    ro babel-metric-comp-algorithms<1..*>;
     string                    ro babel-security-supported<0..*>;
    [string                    ro babel-mac-algorithms<1..*>;]
    [string                    ro babel-dtls-cert-types<1..*>;]
    [boolean                   rw babel-stats-enable;]
    [operation                    babel-stats-reset;]
     babel-constants-obj       ro babel-constants;
     babel-interface-obj       ro babel-interfaces<0..*>;
     babel-route-obj           ro babel-routes<0..*>;
    [babel-mac-key-set-obj     rw babel-mac-key-sets<0..*>;]
    [babel-dtls-cert-set-obj   rw babel-dtls-cert-sets<0..*>;]
} babel-information-obj;
```

babel-implementation-version: The name and version of this implementation of the Babel protocol.

babel-enable: When written, it configures whether the protocol should be enabled (true) or disabled (false). A read from the running or intended datastore indicates the configured administrative value of whether the protocol is enabled (true) or not (false). A read from the operational datastore indicates whether the protocol is actually running (true) or not (i.e., it indicates the operational state of the protocol). A data model that does not replicate parameters for running and operational datastores can implement this as two separate parameters. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-self-router-id: The router-id used by this instance of the Babel protocol to identify itself. [I-D.ietf-babel-rfc6126bis] describes this as an arbitrary string of 8 octets. The router-id value MUST NOT consist of all zeroes or all ones.

babel-self-seqno: The current sequence number included in route updates for routes originated by this node. This is a 16-bit unsigned integer.

babel-metric-comp-algorithms: List of supported cost computation algorithms. Possible values include "2-out-of-3", and "ETX". "2-out-of-3" is described in [I-D.ietf-babel-rfc6126bis], section A.2.1. "ETX" is described in [I-D.ietf-babel-rfc6126bis], section A.2.2.

babel-security-supported: List of supported security mechanisms. Possible values include "MAC" and "DTLS".

babel-mac-algorithms: List of supported MAC computation algorithms. Possible values include "HMAC-SHA256", "BLAKE2s".

babel-dtls-cert-types: List of supported DTLS certificate types. Possible values include "X.509" and "RawPublicKey".

babel-stats-enable: Indicates whether statistics collection is enabled (true) or disabled (false) on all interfaces. When enabled, existing statistics values are not cleared and will be incremented as new packets are counted.

babel-stats-reset: An operation that resets all babel-if-stats parameters to zero. This operation has no input or output parameters.

babel-constants: A babel-constants-obj object.

babel-interfaces: A set of babel-interface-obj objects.

babel-routes: A set of babel-route-obj objects. Contains the routes known to this node.

babel-mac-key-sets: A set of babel-mac-key-set-obj objects. If this object is implemented, it provides access to parameters related to the MAC security mechanism. An implementation MAY choose to expose this object as read-only ("ro").

babel-dtls-cert-sets: A set of babel-dtls-cert-set-obj objects. If this object is implemented, it provides access to parameters related to the DTLS security mechanism. An implementation MAY choose to expose this object as read-only ("ro").

3.2. Definition of babel-constants-obj

```
object {
     uint         rw babel-udp-port;
    [ip-address   rw babel-mcast-group;]
} babel-constants-obj;
```

babel-udp-port: UDP port for sending and listening for Babel packets. Default is 6696. An implementation MAY choose to expose this parameter as read-only ("ro"). This is a 16-bit unsigned integer.

babel-mcast-group: Multicast group for sending and listening to multicast announcements on IPv6. Default is ff02::1:6. An implementation MAY choose to expose this parameter as read-only ("ro").

3.3. Definition of babel-interface-obj

```
object {
     reference            ro babel-interface-reference;
    [boolean              rw babel-interface-enable;]
     string               rw babel-interface-metric-algorithm;
    [boolean              rw babel-interface-split-horizon;]
    [uint                 ro babel-mcast-hello-seqno;]
    [uint                 ro babel-mcast-hello-interval;]
    [uint                 ro babel-update-interval;]
    [boolean              rw babel-mac-enable;]
    [reference            rw babel-if-mac-key-sets<0..*>;]
    [boolean              rw babel-mac-verify;]
    [boolean              rw babel-dtls-enable;]
    [reference            rw babel-if-dtls-cert-sets<0..*>;]
    [boolean              rw babel-dtls-cached-info;]
    [string               rw babel-dtls-cert-prefer<0..*>;]
    [boolean              rw babel-packet-log-enable;]
    [reference            ro babel-packet-log;]
    [babel-if-stats-obj   ro babel-if-stats;]
     babel-neighbor-obj   ro babel-neighbors<0..*>;
} babel-interface-obj;
```

babel-interface-reference: Reference to an interface object that can be used to send and receive IPv6 packets, as defined by the data model (e.g., YANG [RFC7950], BBF [TR-181]). Referencing syntax will be specific to the data model. If there is no set of interface objects available, this should be a string that indicates the interface name used by the underlying operating system.

babel-interface-enable: When written, it configures whether the protocol should be enabled (true) or disabled (false) on this interface. A read from the running or intended datastore indicates the configured administrative value of whether the protocol is enabled (true) or not (false). A read from the operational datastore indicates whether the protocol is actually running (true) or not (i.e., it indicates the operational state of the protocol). A data model that does not replicate parameters for running and operational datastores can implement this as two separate parameters. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-interface-metric-algorithm: Indicates the metric computation algorithm used on this interface. The value MUST be one of those listed in the babel-information-obj babel-metric-comp-algorithms parameter. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-interface-split-horizon: Indicates whether or not the split horizon optimization is used when calculating metrics on this interface. A value of true indicates split horizon optimization is used. Split horizon optimization is described in [I-D.ietf-babel-rfc6126bis], section 3.7.4. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-mcast-hello-seqno: The current sequence number in use for multicast Hellos sent on this interface. This is a 16-bit unsigned integer.

babel-mcast-hello-interval: The current interval in use for multicast Hellos sent on this interface. Units are centiseconds. This is a 16-bit unsigned integer.

babel-update-interval: The current interval in use for all updates (multicast and unicast) sent on this interface. Units are centiseconds. This is a 16-bit unsigned integer.

babel-mac-enable: Indicates whether the MAC security mechanism is enabled (true) or disabled (false). An implementation MAY choose to expose this parameter as read-only ("ro").

babel-if-mac-keys-sets: List of references to the babel-mac entries that apply to this interface. When an interface instance is created, all babel-mac-key-sets instances with babel-mac-default-apply "true" will be included in this list. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-mac-verify A Boolean flag indicating whether MAC hashes in incoming Babel packets are required to be present and are verified. If this parameter is "true", incoming packets are required to have a valid MAC hash. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-dtls-enable: Indicates whether the DTLS security mechanism is enabled (true) or disabled (false). An implementation MAY choose to expose this parameter as read-only ("ro").

babel-if-dtls-cert-sets: List of references to the babel-dtls-cert-sets entries that apply to this interface. When an interface instance is created, all babel-dtls-cert-sets instances with babel-dtls-default-apply "true" will be included in this list. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-dtls-cached-info: Indicates whether the cached_info extension is included in ClientHello and ServerHello packets. The extension is included if the value is "true". An implementation MAY choose to expose this parameter as read-only ("ro").

babel-dtls-cert-prefer: List of supported certificate types, in order of preference. The values MUST be among those listed in the babel-dtls-cert-types parameter. This list is used to populate the server_certificate_type extension in a Client Hello. Values that are present in at least one instance in the babel-dtls-certs object of a referenced babel-dtls instance and that have a non-empty babel-cert-private-key will be used to populate the client_certificate_type extension in a Client Hello.

babel-packet-log-enable: Indicates whether packet logging is enabled (true) or disabled (false) on this interface.

babel-packet-log: A reference or url link to a file that contains a timestamped log of packets received and sent on babel-udp-port on this interface. The [libpcap] file format with .pcap file extension SHOULD be supported for packet log files. Logging is enabled / disabled by babel-packet-log-enable. Implementations will need to carefully manage and limit memory used by packet logs.

babel-if-stats: Statistics collection object for this interface.

babel-neighbors: A set of babel-neighbor-obj objects.

3.4. Definition of babel-if-stats-obj

```
object {
     uint   ro babel-sent-mcast-hello;
     uint   ro babel-sent-mcast-update;
     uint   ro babel-sent-ucast-hello;
     uint   ro babel-sent-ucast-update;
     uint   ro babel-sent-IHU;
     uint   ro babel-received-packets;
} babel-if-stats-obj;
```

babel-sent-mcast-hello: A count of the number of multicast Hello packets sent on this interface.

babel-sent-mcast-update: A count of the number of multicast update packets sent on this interface.

babel-sent-ucast-hello: A count of the number of unicast Hello packets sent on this interface.

babel-sent-ucast-update: A count of the number of unicast update packets sent on this interface.

babel-sent-IHU: A count of the number of IHU packets sent on this interface.

babel-received-packets: A count of the number of Babel packets received on this interface.

3.5. Definition of babel-neighbor-obj

```
object {
     ip-address   ro babel-neighbor-address;
    [binary       ro babel-hello-mcast-history;]
    [binary       ro babel-hello-ucast-history;]
     uint         ro babel-txcost;
     uint         ro babel-exp-mcast-hello-seqno;
     uint         ro babel-exp-ucast-hello-seqno;
    [uint         ro babel-ucast-hello-seqno;]
    [uint         ro babel-ucast-hello-interval;]
    [uint         ro babel-rxcost;]
    [uint         ro babel-cost;]
} babel-neighbor-obj;
```

babel-neighbor-address: IPv4 or IPv6 address the neighbor sends packets from.

babel-hello-mcast-history: The multicast Hello history of whether or not the multicast Hello packets prior to babel-exp-mcast-hello-seqno were received. A binary sequence where the most recently received Hello is expressed as a "1" placed in the left-most bit, with prior bits shifted right (and "0" bits placed between prior Hello bits and most recent Hello for any not-received Hellos). This value should be displayed using hex digits ([0-9a-fA-F]). See [I-D.ietf-babel-rfc6126bis], section A.1.

babel-hello-ucast-history: The unicast Hello history of whether or not the unicast Hello packets prior to babel-exp-ucast-hello-seqno were received. A binary sequence where the most recently received Hello is expressed as a "1" placed in the left-most bit, with prior bits shifted right (and "0" bits placed between prior Hello bits and most recent Hello for any not-received Hellos). This value should be displayed using hex digits ([0-9a-fA-F]). See [I-D.ietf-babel-rfc6126bis], section A.1.

babel-txcost: Transmission cost value from the last IHU packet received from this neighbor, or maximum value to indicate the IHU hold timer for this neighbor has expired. See [I-D.ietf-babel-rfc6126bis], section 3.4.2. This is a 16-bit unsigned integer.

babel-exp-mcast-hello-seqno: Expected multicast Hello sequence number of next Hello to be received from this neighbor. If multicast Hello packets are not expected, or processing of multicast packets is not enabled, this MUST be NULL. This is a 16-bit unsigned integer; if the data model uses zero (0) to represent NULL values for unsigned integers, the data model MAY use a different data type that allows differentiation between zero (0) and NULL.

babel-exp-ucast-hello-seqno: Expected unicast Hello sequence number of next Hello to be received from this neighbor. If unicast Hello packets are not expected, or processing of unicast packets is not enabled, this MUST be NULL. This is a 16-bit unsigned integer; if the data model uses zero (0) to represent NULL values for unsigned integers, the data model MAY use a different data type that allows differentiation between zero (0) and NULL.

babel-ucast-hello-seqno: The current sequence number in use for unicast Hellos sent to this neighbor. If unicast Hellos are not being sent, this MUST be NULL. This is a 16-bit unsigned integer; if the data model uses zero (0) to represent NULL values for unsigned integers, the data model MAY use a different data type that allows differentiation between zero (0) and NULL.

babel-ucast-hello-interval: The current interval in use for unicast Hellos sent to this neighbor. Units are centiseconds. This is a 16-bit unsigned integer.

babel-rxcost: Reception cost calculated for this neighbor. This value is usually derived from the Hello history, which may be combined with other data, such as statistics maintained by the link layer. The rxcost is sent to a neighbor in each IHU. See [I-D.ietf-babel-rfc6126bis], section 3.4.3. This is a 16-bit unsigned integer.

babel-cost: The link cost, as computed from the values maintained in the neighbor table: the statistics kept in the neighbor table about the reception of Hellos, and the txcost computed from received IHU packets. This is a 16-bit unsigned integer.

3.6. Definition of babel-route-obj

```
object {
     ip-address   ro babel-route-prefix;
     uint         ro babel-route-prefix-length;
     binary       ro babel-route-router-id;
     string       ro babel-route-neighbor;
     uint         ro babel-route-received-metric;
     uint         ro babel-route-calculated-metric;
     uint         ro babel-route-seqno;
     ip-address   ro babel-route-next-hop;
     boolean      ro babel-route-feasible;
     boolean      ro babel-route-selected;
} babel-route-obj;
```

babel-route-prefix: Prefix (expressed in IP address format) for which this route is advertised.

babel-route-prefix-length: Length of the prefix for which this route is advertised.

babel-route-router-id: The router-id of the router that originated this route.

babel-route-neighbor: Reference to the babel-neighbors entry for the neighbor that advertised this route.

babel-route-received-metric: The metric with which this route was advertised by the neighbor, or maximum value to indicate the route was recently retracted and is temporarily unreachable (see Section 3.5.5 of [I-D.ietf-babel-rfc6126bis]). This metric will be NULL if the route was not received from a neighbor but was

generated through other means. At least one of babel-route-calculated-metric and babel-route-received-metric MUST be non-NULL. Having both be non-NULL is expected for a route that is received and subsequently advertised. This is a 16-bit unsigned integer; if the data model uses zero (0) to represent NULL values for unsigned integers, the data model MAY use a different data type that allows differentiation between zero (0) and NULL.

babel-route-calculated-metric: A calculated metric for this route. How the metric is calculated is implementation-specific. Maximum value indicates the route was recently retracted and is temporarily unreachable (see Section 3.5.5 of [I-D.ietf-babel-rfc6126bis]). At least one of babel-route-calculated-metric and babel-route-received-metric MUST be non-NULL. Having both be non-NULL is expected for a route that is received and subsequently advertised. This is a 16-bit unsigned integer; if the data model uses zero (0) to represent NULL values for unsigned integers, the data model MAY use a different data type that allows differentiation between zero (0) and NULL.

babel-route-seqno: The sequence number with which this route was advertised. This is a 16-bit unsigned integer.

babel-route-next-hop: The next-hop address of this route. This will be empty if this route has no next-hop address.

babel-route-feasible: A Boolean flag indicating whether this route is feasible, as defined in Section 3.5.1 of [I-D.ietf-babel-rfc6126bis]).

babel-route-selected: A Boolean flag indicating whether this route is selected (i.e., whether it is currently being used for forwarding and is being advertised).

3.7. Definition of babel-mac-key-set-obj

```
object {
     boolean             rw babel-mac-default-apply;
     babel-mac-key-obj   rw babel-mac-keys<0..*>;
} babel-mac-obj;
```

babel-mac-default-apply: A Boolean flag indicating whether this babel-mac instance is applied to all new babel-interface instances, by default. If "true", this instance is applied to new babel-interfaces instances at the time they are created, by including it in the babel-interface-mac-keys list. If "false", this instance is not applied to new babel-interfaces instances

when they are created. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-mac-keys: A set of babel-mac-key-obj objects.

3.8. Definition of babel-mac-key-obj

```
object {
     string      rw babel-mac-key-name;
     boolean     rw babel-mac-key-use-sign;
     boolean     rw babel-mac-key-use-verify;
     binary      -- babel-mac-key-value;
     string      rw babel-mac-key-algorithm;
    [operation      babel-mac-key-test;]
} babel-mac-key-obj;
```

babel-mac-key-name: A unique name for this MAC key that can be used to identify the key in this object instance, since the key value is not allowed to be read. This value MUST NOT be empty and can only be provided when this instance is created (i.e., it is not subsequently writable). The value MAY be auto-generated if not explicitly supplied when the instance is created.

babel-mac-key-use-sign: Indicates whether this key value is used to sign sent Babel packets. Sent packets are signed using this key if the value is "true". If the value is "false", this key is not used to sign sent Babel packets. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-mac-key-use-verify: Indicates whether this key value is used to verify incoming Babel packets. This key is used to verify incoming packets if the value is "true". If the value is "false", no MAC is computed from this key for comparing with the MAC in an incoming packet. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-mac-key-value: The value of the MAC key. An implementation MUST NOT allow this parameter to be read. This can be done by always providing an empty string when read, or through permissions, or other means. This value MUST be provided when this instance is created, and is not subsequently writable. This value is of a length suitable for the associated babel-mac-key-algorithm. If the algorithm is based on the HMAC construction [RFC2104], the length MUST be between 0 and the block size of the underlying hash inclusive (where "HMAC-SHA256" block size is 64 bytes as described in [RFC4868]). If the algorithm is "BLAKE2s", the length MUST be between 0 and 32 bytes inclusive, as described in [RFC7693].

babel-mac-key-algorithm The name of the MAC algorithm used with this key. The value MUST be the same as one of the enumerations listed in the babel-mac-algorithms parameter. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-mac-key-test: An operation that allows the MAC key and hash algorithm to be tested to see if they produce an expected outcome. Input to this operation is a binary string. The implementation is expected to create a hash of this string using the babel-mac-key-value and the babel-mac-key-algorithm. The output of this operation is the resulting hash, as a binary string.

3.9. Definition of babel-dtls-cert-set-obj

```
object {
     boolean               rw babel-dtls-default-apply;
     babel-dtls-cert-obj   rw babel-dtls-certs<0..*>;
} babel-dtls-cert-set-obj;
```

babel-dtls-default-apply: A Boolean flag indicating whether this babel-dtls instance is applied to all new babel-interface instances, by default. If "true", this instance is applied to new babel-interfaces instances at the time they are created, by including it in the babel-interface-dtls-certs list. If "false", this instance is not applied to new babel-interfaces instances when they are created. An implementation MAY choose to expose this parameter as read-only ("ro").

babel-dtls-certs: A set of babel-dtls-cert-obj objects. This contains both certificates for this implementation to present for authentication, and to accept from others. Certificates with a non-empty babel-cert-private-key can be presented by this implementation for authentication.

3.10. Definition of babel-dtls-cert-obj

```
object {
     string      rw babel-cert-name;
     string      rw babel-cert-value;
     string      rw babel-cert-type;
     binary      -- babel-cert-private-key;
    [operation      babel-cert-test;]
} babel-dtls-cert-obj;
```

babel-cert-name: A unique name for this DTLS certificate that can be used to identify the certificate in this object instance, since the value is too long to be useful for identification. This value MUST NOT be empty and can only be provided when this instance is

created (i.e., it is not subsequently writable). The value MAY be auto-generated if not explicitly supplied when the instance is created.

babel-cert-value: The DTLS certificate in PEM format [RFC7468]. This value MUST be provided when this instance is created, and is not subsequently writable.

babel-cert-type: The name of the certificate type of this object instance. The value MUST be the same as one of the enumerations listed in the babel-dtls-cert-types parameter. This value can only be provided when this instance is created, and is not subsequently writable.

babel-cert-private-key: The value of the private key. If this is non-empty, this certificate can be used by this implementation to provide a certificate during DTLS handshaking. An implementation MUST NOT allow this parameter to be read. This can be done by always providing an empty string when read, or through permissions, or other means. This value can only be provided when this instance is created, and is not subsequently writable.

babel-cert-test: An operation that allows a hash of the provided input string to be created using the certificate public key and the SHA-256 hash algorithm. Input to this operation is a binary string. The output of this operation is the resulting hash, as a binary string.

4. Extending the Information Model

Implementations MAY extend this information model with other parameters or objects. For example, an implementation MAY choose to expose Babel route filtering rules by adding a route filtering object with parameters appropriate to how route filtering is done in that implementation. The precise means used to extend the information model would be specific to the data model the implementation uses to expose this information.

5. Security Considerations

This document defines a set of information model objects and parameters that may be exposed to be visible from other devices, and some of which may be configured. Securing access to and ensuring the integrity of this data is in scope of and the responsibility of any data model derived from this information model. Specifically, any YANG [RFC7950] data model is expected to define security exposure of the various parameters, and a [TR-181] data model will be secured by

the mechanisms defined for the management protocol used to transport it.

Misconfiguration (whether unintentional or malicious) can prevent reachability or cause poor network performance (increased latency, jitter, etc.). The information in this model discloses network topology, which can be used to mount subsequent attacks on traffic traversing the network.

This information model defines objects that can allow credentials (for this device, for trusted devices, and for trusted certificate authorities) to be added and deleted. Public keys may be exposed through this model. This model requires that private keys never be exposed. The Babel security mechanisms that make use of these credentials (e.g., [I-D.ietf-babel-dtls], [I-D.ietf-babel-hmac]) identify what credentials can be used with those mechanisms.

MAC keys are allowed to be as short as zero-length. This is useful for testing. Network operators are advised to follow current best practices for key length and generation of keys related to the MAC algorithm associated with the key. Short (and zero-length) keys and keys that make use of only alphanumeric characters are highly susceptible to brute force attacks.

6. IANA Considerations

This document has no IANA actions.

7. Acknowledgements

Juliusz Chroboczek, Toke Hoeiland-Joergensen, David Schinazi, Antonin Decimo, Acee Lindem, and Carsten Bormann have been very helpful in refining this information model.

The language in the Notation section was mostly taken from [RFC8193].

8. References

8.1. Normative References

[I-D.ietf-babel-rfc6126bis]
Chroboczek, J. and D. Schinazi, "The Babel Routing Protocol", draft-ietf-babel-rfc6126bis-19 (work in progress), August 2020.

[libpcap] Wireshark, "Libpcap File Format", 2015, <https://wiki.wireshark.org/Development/LibpcapFileFormat>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

8.2. Informative References

[I-D.ietf-babel-dtls] Decimo, A., Schinazi, D., and J. Chroboczek, "Babel Routing Protocol over Datagram Transport Layer Security", draft-ietf-babel-dtls-10 (work in progress), June 2020.

[I-D.ietf-babel-hmac] Do, C., Kolodziejak, W., and J. Chroboczek, "MAC authentication for the Babel routing protocol", draft-ietf-babel-hmac-10 (work in progress), August 2019.

[ISO.10646] International Organization for Standardization, "Information Technology - Universal Multiple-Octet Coded Character Set (UCS)", ISO Standard 10646:2014, 2014.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC4868] Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec", RFC 4868, DOI 10.17487/RFC4868, May 2007, <https://www.rfc-editor.org/info/rfc4868>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/info/rfc6241>.

[RFC7693] Saarinen, M-J., Ed. and J-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFC 7693, DOI 10.17487/RFC7693, November 2015, <https://www.rfc-editor.org/info/rfc7693>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8193] Burbridge, T., Eardley, P., Bagnulo, M., and J. Schoenwaelder, "Information Model for Large-Scale Measurement Platforms (LMAPs)", RFC 8193, DOI 10.17487/RFC8193, August 2017, <https://www.rfc-editor.org/info/rfc8193>.

[TR-181] Broadband Forum, "Device Data Model", <http://cwmp-data-models.broadband-forum.org/>.

Authors' Addresses

Barbara Stark
AT&T
Atlanta, GA
US

Email: barbara.stark@att.com

Mahesh Jethanandani
VMware
California
US

Email: mjethanandani@gmail.com

Network Working Group
Internet-Draft
Obsoletes: 6126,7557 (if approved)
Intended status: Standards Track
Expires: April 26, 2019

J. Chroboczek
IRIF, University of Paris-Diderot
D. Schinazi
Apple Inc.

October 23, 2018

The Babel Routing Protocol
draft-ietf-babel-rfc6126bis-06

Abstract

Babel is a loop-avoiding distance-vector routing protocol that is robust and efficient both in ordinary wired networks and in wireless mesh networks. This document describes the Babel routing protocol, and obsoletes RFCs 6126 and 7557

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 26, 2019.

Copyright Notice

Table of Contents

1. Introduction

Babel is a loop-avoiding distance-vector routing protocol that is designed to be robust and efficient both in networks using prefix-based routing and in networks using flat routing ("mesh networks"), and both in relatively stable wired networks and in highly dynamic wireless networks.

1.1. Features

The main property that makes Babel suitable for unstable networks is that, unlike naive distance-vector routing protocols [RIP], it strongly limits the frequency and duration of routing pathologies such as routing loops and black-holes during reconvergence. Even after a mobility event is detected, a Babel network usually remains loop-free. Babel then quickly reconverges to a configuration that preserves the loop-freedom and connectedness of the network, but is not necessarily optimal; in many cases, this operation requires no packet exchanges at all. Babel then slowly converges, in a time on the scale of minutes, to an optimal configuration. This is achieved by using sequenced routes, a technique pioneered by Destination-Sequenced Distance-Vector routing [DSDV].

More precisely, Babel has the following properties:

- when every prefix is originated by at most one router, Babel never suffers from routing loops;

- when a single prefix is originated by multiple routers, Babel may occasionally create a transient routing loop for this particular prefix; this loop disappears in a time proportional to its diameter, and never again (up to an arbitrary garbage-collection (GC) time) will the routers involved participate in a routing loop for the same prefix;

- assuming bounded packet loss rates, any routing black-holes that may appear after a mobility event are corrected in a time at most proportional to the network's diameter.

Babel has provisions for link quality estimation and for fairly arbitrary metrics. When configured suitably, Babel can implement shortest-path routing, or it may use a metric based, for example, on measured packet loss.

Babel nodes will successfully establish an association even when they are configured with different parameters. For example, a mobile node that is low on battery may choose to use larger time constants (hello and update intervals, etc.) than a node that has access to wall power. Conversely, a node that detects high levels of mobility may choose to use smaller time constants. The ability to build such heterogeneous networks makes Babel particularly adapted to the unmanaged and wireless environment.

Finally, Babel is a hybrid routing protocol, in the sense that it can carry routes for multiple network-layer protocols (IPv4 and IPv6), whichever protocol the Babel packets are themselves being carried over.

1.2. Limitations

Babel has two limitations that make it unsuitable for use in some environments. First, Babel relies on periodic routing table updates rather than using a reliable transport; hence, in large, stable networks it generates more traffic than protocols that only send updates when the network topology changes. In such networks, protocols such as OSPF [OSPF], IS-IS [IS-IS], or the Enhanced Interior Gateway Routing Protocol (EIGRP) [EIGRP] might be more suitable.

Second, unless the optional algorithm described in Section 3.5.5 is implemented, Babel does impose a hold time when a prefix is retracted. While this hold time does not apply to the exact prefix being retracted, and hence does not prevent fast reconvergence should it become available again, it does apply to any shorter prefix that covers it. This may make those implementations of Babel that do not implement the optional algorithm described in Section 3.5.5 unsuitable for use in networks that implement automatic prefix aggregation.

1.3. Specification of Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Conceptual Description of the Protocol

Babel is a loop-avoiding distance vector protocol: it is based on the Bellman-Ford protocol, just like the venerable RIP [RIP], but includes a number of refinements that either prevent loop formation altogether, or ensure that a loop disappears in a timely manner and doesn't form again.

Conceptually, Bellman-Ford is executed in parallel for every source of routing information (destination of data traffic). In the following discussion, we fix a source S; the reader will recall that the same algorithm is executed for all sources.

2.1. Costs, Metrics and Neighbourship

For every pair of neighbouring nodes A and B, Babel computes an abstract value known as the cost of the link from A to B., written C(A, B). Given a route between any two (not necessarily neighbouring) nodes, the metric of the route is the sum of the costs of all the edges along the route. The goal of the routing algorithm is to compute, for every source S, the tree of routes of lowest metric to S.

Costs and metrics need not be integers. In general, they can be values in any algebra that satisfies two fairly general conditions (Section 3.5.2).

A Babel node periodically sends Hello messages to all of its neighbours; it also periodically sends an IHU ("I Heard You") message to every neighbour from which it has recently heard a Hello. From the information derived from Hello and IHU messages received from its neighbour B, a node A computes the cost C(A, B) of the link from A to B.

2.2. The Bellman-Ford Algorithm

Every node A maintains two pieces of data: its estimated distance to S, written D(A), and its next-hop router to S, written NH(A). Initially, D(S) = 0, D(A) is infinite, and NH(A) is undefined.

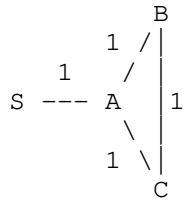
Periodically, every node B sends to all of its neighbours a route update, a message containing D(B). When a neighbour A of B receives the route update, it checks whether B is its selected next hop; if that is the case, then NH(A) is set to B, and D(A) is set to C(A, B) + D(B). If that is not the case, then A compares C(A, B) + D(B) to its current value of D(A). If that value is smaller, meaning that the received update advertises a route that is better than the

currently selected route, then NH(A) is set to B, and D(A) is set to C(A, B) + D(B).

A number of refinements to this algorithm are possible, and are used by Babel. In particular, convergence speed may be increased by sending unscheduled "triggered updates" whenever a major change in the topology is detected, in addition to the regular, scheduled updates. Additionally, a node may maintain a number of alternate routes, which are being advertised by neighbours other than its selected neighbour, and which can be used immediately if the selected route were to fail.

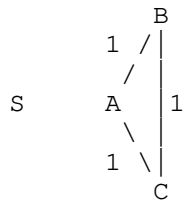
2.3. Transient Loops in Bellman-Ford

It is well known that a naive application of Bellman-Ford to distributed routing can cause transient loops after a topology change. Consider for example the following topology:



After convergence, D(B) = D(C) = 2, with NH(B) = NH(C) = A.

Suppose now that the link between S and A fails:



When it detects the failure of the link, A switches its next hop to B (which is still advertising a route to S with metric 2), and advertises a metric equal to 3, and then advertises a new route with metric 3. This process of nodes changing selected neighbours and increasing their metric continues until the advertised metric reaches "infinity", a value larger than all the metrics that the routing protocol is able to carry.

2.4. Feasibility Conditions

Bellman-Ford is a very robust algorithm: its convergence properties are preserved when routers delay route acquisition or when they discard some updates. Babel routers discard received route announcements unless they can prove that accepting them cannot possibly cause a routing loop.

More formally, we define a condition over route announcements, known as the "feasibility condition", that guarantees the absence of routing loops whenever all routers ignore route updates that do not satisfy the feasibility condition. In effect, this makes Bellman-Ford into a family of routing algorithms, parameterised by the feasibility condition.

Many different feasibility conditions are possible. For example, BGP can be modelled as being a distance-vector protocol with a (rather drastic) feasibility condition: a routing update is only accepted when the receiving node's AS number is not included in the update's AS-Path attribute (note that BGP's feasibility condition does not ensure the absence of transient "micro-loops" during reconvergence).

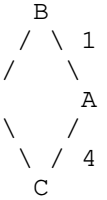
Another simple feasibility condition, used in the Destination-Sequenced Distance-Vector (DSDV) routing protocol [DSDV] and in the Ad hoc On-Demand Distance Vector (AODV) protocol, stems from the following observation: a routing loop can only arise after a router has switched to a route with a larger metric than the route that it had previously selected. Hence, one could decide that a route is feasible only when its metric at the local node would be no larger than the metric of the currently selected route, i.e., an announcement carrying a metric D(B) is accepted by A when C(A, B) + D(B) <= D(A). If all routers obey this constraint, then the metric at every router is nonincreasing, and the following invariant is always preserved: if A has selected B as its successor, then D(B) < D(A), which implies that the forwarding graph is loop-free.

Babel uses a slightly more refined feasibility condition, derived from EIGRP [DUAL]. Given a router A, define the feasibility distance of A, written FD(A), as the smallest metric that A has ever advertised for S to any of its neighbours. An update sent by a neighbour B of A is feasible when the metric D(B) advertised by B is strictly smaller than A's feasibility distance, i.e., when D(B) < FD(A).

It is easy to see that this latter condition is no more restrictive than DSDV-feasibility. Suppose that node A obeys DSDV-feasibility; then D(A) is nonincreasing, hence at all times D(A) <= FD(A). Suppose now that A receives a DSDV-feasible update that advertises a

metric D(B). Since the update is DSDV-feasible, C(A, B) + D(B) <= D(A), hence D(B) < D(A), and since D(A) <= FD(A), D(B) < FD(A).

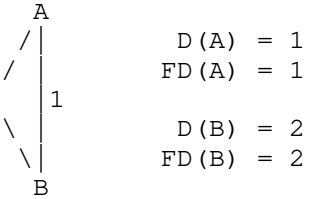
To see that it is strictly less restrictive, consider the following diagram, where A has selected the route through B, and D(A) = FD(A) = 2. Since D(C) = 1 < FD(A), the alternate route through C is feasible for A, although its metric C(A, C) + D(C) = 5 is larger than that of the currently selected route:



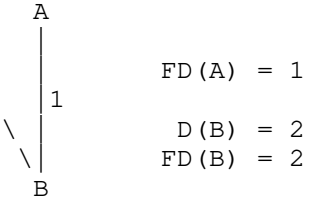
To show that this feasibility condition still guarantees loop-freedom, recall that at the time when A accepts an update from B, the metric D(B) announced by B is no smaller than FD(B); since it is smaller than FD(A), at that point in time FD(B) < FD(A). Since this property is preserved when A sends updates, it remains true at all times, which ensures that the forwarding graph has no loops.

2.5. Solving Starvation: Sequencing Routes

Obviously, the feasibility conditions defined above cause starvation when a router runs out of feasible routes. Consider the following diagram, where both A and B have selected the direct route to S:



Suppose now that the link between A and S breaks:

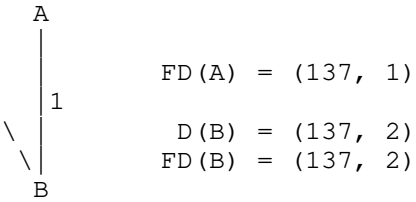


The only route available from A to S, the one that goes through B, is not feasible: A suffers from spurious starvation. At that point, the whole subtree suffering from starvation must be reset, which is essentially what EIGRP does when it performs a global synchronisation of all the routers in the sarving subtree (the "active" phase of EIGRP).

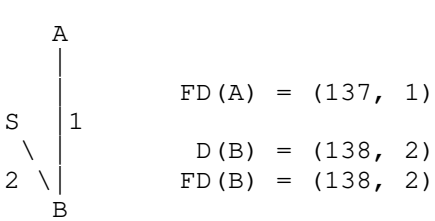
Babel reacts to starvation in a less drastic manner, by using sequenced routes, a technique introduced by DSDV and adopted by AODV. In addition to a metric, every route carries a sequence number, a nondecreasing integer that is propagated unchanged through the network and is only ever incremented by the source; a pair (s, m), where s is a sequence number and m a metric, is called a distance.

A received update is feasible when either it is more recent than the feasibility distance maintained by the receiving node, or it is equally recent and the metric is strictly smaller. More formally, if FD(A) = (s, m), then an update carrying the distance (s', m') is feasible when either s' > s, or s = s' and m' < m.

Assuming the sequence number of S is 137, the diagram above becomes:



After S increases its sequence number, and the new sequence number is propagated to B, we have:



at which point the route through B becomes feasible again.

Note that while sequence numbers are used for determining feasibility, they are not necessarily used in route selection: a node will normally ignore the sequence number when selecting the best route to a given destination (Section 3.6).

2.6. Requests

In DSDV, the sequence number of a source is increased periodically. A route becomes feasible again after the source increases its sequence number, and the new sequence number is propagated through the network, which may, in general, require a significant amount of time.

Babel takes a different approach. When a node detects that it is suffering from a potentially spurious starvation, it sends an explicit request to the source for a new sequence number. This request is forwarded hop by hop to the source, with no regard to the feasibility condition. Upon receiving the request, the source increases its sequence number and broadcasts an update, which is forwarded to the requesting node.

Note that after a change in network topology not all such requests will, in general, reach the source, as some will be sent over links that are now broken. However, if the network is still connected, then at least one among the nodes suffering from spurious starvation has an (unfeasible) route to the source; hence, in the absence of packet loss, at least one such request will reach the source. (Resending requests a small number of times compensates for packet loss.)

Since requests are forwarded with no regard to the feasibility condition, they may, in general, be caught in a forwarding loop; this is avoided by having nodes perform duplicate detection for the requests that they forward.

2.7. Multiple Routers

The above discussion assumes that every prefix is originated by a single router. In real networks, however, it is often necessary to have a single prefix originated by multiple routers: for example, the default route will be originated by all of the edge routers of a routing domain.

Since synchronising sequence numbers between distinct routers is problematic, Babel treats routes for the same prefix as distinct entities when they are originated by different routers: every route announcement carries the router-id of its originating router, and feasibility distances are not maintained per prefix, but per source, where a source is a pair of a router-id and a prefix. In effect, Babel guarantees loop-freedom for the forwarding graph to every source; since the union of multiple acyclic graphs is not in general acyclic, Babel does not in general guarantee loop-freedom when a prefix is originated by multiple routers, but any loops will be

broken in a time at most proportional to the diameter of the loop -- as soon as an update has "gone around" the routing loop.

Consider for example the following topology, where A has selected the default route through S, and B has selected the one through S':

```
           1     1     1
::/0 -- S --- A --- B --- S' -- ::/0
```

Suppose that both default routes fail at the same time; then nothing prevents A from switching to B, and B simultaneously switching to A. However, as soon as A has successfully advertised the new route to B, the route through A will become unfeasible for B. Conversely, as soon as B will have advertised the route through A, the route through B will become unfeasible for A.

In effect, the routing loop disappears at the latest when routing information has gone around the loop. Since this process can be delayed by lost packets, Babel makes certain efforts to ensure that updates are sent reliably after a router-id change (Section 3.7.2).

Additionally, after the routers have advertised the two routes, both sources will be in their source tables, which will prevent them from ever again participating in a routing loop involving routes from S and S' (up to the source GC time, which, available memory permitting, can be set to arbitrarily large values).

2.8. Overlapping Prefixes

In the above discussion, we have assumed that all prefixes are disjoint, as is the case in flat ("mesh") routing. In practice, however, prefixes may overlap: for example, the default route overlaps with all of the routes present in the network.

After a route fails, it is not correct in general to switch to a route that subsumes the failed route. Consider for example the following configuration:

```
           1     1
::/0 -- A --- B --- C
```

Suppose that node C fails. If B forwards packets destined to C by following the default route, a routing loop will form, and persist until A learns of B's retraction of the direct route to C. B avoids this pitfall by installing an "unreachable" route after a route is retracted; this route is maintained until it can be guaranteed that the former route has been retracted by all of B's neighbours (Section 3.5.5).

3. Protocol Operation

Every Babel speaker is assigned a router-id, which is an arbitrary string of 8 octets that is assumed unique across the routing domain. For example, routers-ids could be assigned randomly, or they could derived from a link-layer address. (The protocol encoding is slightly more compact when router-ids are assigned in the same manner as the IPv6 layer assigns host IDs.)

3.1. Message Transmission and Reception

Babel protocol packets are sent in the body of a UDP datagram (as described in Section 4 below). Each Babel packet consists of zero or more TLVs. Most TLVs may contain sub-TLVs.

The source address of a Babel packet is always a unicast address, link-local in the case of IPv6. Babel packets may be sent to a well-known (link-local) multicast address or to a (link-local) unicast address. In normal operation, a Babel speaker sends both multicast and unicast packets to its neighbours.

With the exception of Hello TLVs and acknowledgments, all Babel TLVs can be sent to either unicast or multicast addresses, and their semantics does not depend on whether the destination is a unicast or a multicast address. Hence, a Babel speaker does not need to determine the destination address of a packet that it receives in order to interpret it.

A moderate amount of jitter may be applied to packets sent by a Babel speaker: outgoing TLVs are buffered and SHOULD be sent with a small random delay. This is done for two purposes: it avoids synchronisation of multiple Babel speakers across a network [JITTER], and it allows for the aggregation of multiple TLVs into a single packet.

The exact delay and amount of jitter applied to a packet depends on whether it contains any urgent TLVs. Acknowledgment TLVs MUST be sent before the deadline specified in the corresponding request. The particular class of updates specified in Section 3.7.2 MUST be sent in a timely manner. The particular class of request and update TLVs specified in Section 3.8.2 SHOULD be sent in a timely manner.

3.2. Data Structures

In this section, we give a description of the data structures that every Babel speaker maintains. This description is conceptual: a Babel speaker may use different data structures as long as the resulting protocol is the same as the one described in this document.

For example, rather than maintaining a single table containing both selected and unselected (fallback) routes, as described in Section 3.2.6 belong, an actual implementation would probably use two tables, one with selected routes and one with fallback routes.

3.2.1. Sequence number arithmetic

Sequence numbers (seqnos) appear in a number of Babel data structures, and they are interpreted as integers modulo 2^16. For the purposes of this document, arithmetic on sequence numbers is defined as follows.

Given a seqno s and an integer n, the sum of s and n is defined by

```
s + n (modulo 2^16) = (s + n) MOD 2^16
```

or, equivalently,

```
s + n (modulo 2^16) = (s + n) AND 65535
```

where MOD is the modulo operation yielding a non-negative integer and AND is the bitwise conjunction operation.

Given two sequence numbers s and s', the relation s is less than s' (s < s') is defined by

```
s < s' (modulo 2^16) when 0 < ((s' - s) MOD 2^16) < 32768
```

or equivalently

```
s < s' (modulo 2^16) when s /= s' and ((s' - s) AND 32768) = 0.
```

3.2.2. Node Sequence Number

A node's sequence number is a 16-bit integer that is included in route updates sent for routes originated by this node.

A node increments its sequence number (modulo 2^16) whenever it receives a request for a new sequence number (Section 3.8.1.2). A node SHOULD NOT increment its sequence number (seqno) spontaneously, since increasing seqnos makes it less likely that other nodes will have feasible alternate routes when their selected routes fail.

3.2.3. The Interface Table

The interface table contains the list of interfaces on which the node speaks the Babel protocol. Every interface table entry contains the interface's outgoing Multicast Hello seqno, a 16-bit integer that is

sent with each Multicast Hello TLV on this interface and is incremented (modulo 2^16) whenever a Multicast Hello is sent. (Note that an interface's Multicast Hello seqno is unrelated to the node's seqno.)

There are two timers associated with each interface table entry -- the multicast hello timer, which governs the sending of scheduled Multicast Hello and IHU packets, and the update timer, which governs the sending of periodic route updates.

3.2.4. The Neighbour Table

The neighbour table contains the list of all neighbouring interfaces from which a Babel packet has been recently received. The neighbour table is indexed by pairs of the form (interface, address), and every neighbour table entry contains the following data:

- the local node's interface over which this neighbour is reachable;
- the address of the neighbouring interface;
- a history of recently received Multicast Hello packets from this neighbour; this can, for example, be a sequence of n bits, for some small value n, indicating which of the n hellos most recently sent by this neighbour have been received by the local node;
- a history of recently received Unicast Hello packets from this neighbour;
- the "transmission cost" value from the last IHU packet received from this neighbour, or FFFF hexadecimal (infinity) if the IHU hold timer for this neighbour has expired;
- the neighbour's expected incoming Multicast Hello sequence number, an integer modulo 2^16.
- the neighbour's expected incoming Unicast Hello sequence number, an integer modulo 2^16.
- the neighbour's outgoing Unicast Hello sequence number, an integer modulo 2^16 that is sent with each Unicast Hello TLV to this neighbour and is incremented (modulo 2^16) whenever a Unicast Hello is sent. (Note that a neighbour's outgoing Unicast Hello seqno is distinct from the interface's outgoing Multicast Hello seqno.)

There are three timers associated with each neighbour entry -- the multicast hello timer, which is initialised from the interval value

carried by scheduled Multicast Hello TLVs, the unicast hello timer, which is initialised from the interval value carried by scheduled Unicast Hello TLVs, and the IHU timer, which is initialised to a small multiple of the interval carried in IHU TLVs.

Note that the neighbour table is indexed by IP addresses, not by router-ids: neighbourship is a relationship between interfaces, not between nodes. Therefore, two nodes with multiple interfaces can participate in multiple neighbourship relationships, a situation that can notably arise when wireless nodes with multiple radios are involved.

3.2.5. The Source Table

The source table is used to record feasibility distances. It is indexed by triples of the form (prefix, plen, router-id), and every source table entry contains the following data:

- the prefix (prefix, plen), where plen is the prefix length, that this entry applies to;
- the router-id of a router originating this prefix;
- a pair (seqno, metric), this source's feasibility distance.

There is one timer associated with each entry in the source table -- the source garbage-collection timer. It is initialised to a time on the order of minutes and reset as specified in Section 3.7.3.

3.2.6. The Route Table

The route table contains the routes known to this node. It is indexed by triples of the form (prefix, plen, neighbour), and every route table entry contains the following data:

- the source (prefix, plen, router-id) for which this route is advertised;
- the neighbour that advertised this route;
- the metric with which this route was advertised by the neighbour, or FFFF hexadecimal (infinity) for a recently retracted route;
- the sequence number with which this route was advertised;
- the next-hop address of this route;

- a boolean flag indicating whether this route is selected, i.e., whether it is currently being used for forwarding and is being advertised.

There is one timer associated with each route table entry -- the route expiry timer. It is initialised and reset as specified in Section 3.5.4.

Note that there are two distinct (seqno, metric) pairs associated to each route: the route's distance, which is stored in the route table, and the feasibility distance, stored in the source table and shared between all routes with the same source.

3.2.7. The Table of Pending Seqno Requests

The table of pending seqno requests contains a list of seqno requests that the local node has sent (either because they have been originated locally, or because they were forwarded) and to which no reply has been received yet. This table is indexed by triples of the form (prefix, plen, router-id), and every entry in this table contains the following data:

- the prefix, router-id, and seqno being requested;
- the neighbour, if any, on behalf of which we are forwarding this request;
- a small integer indicating the number of times that this request will be resent if it remains unsatisfied.

There is one timer associated with each pending seqno request; it governs both the resending of requests and their expiry.

3.3. Acknowledgments and acknowledgment requests

A Babel speaker may request that a neighbour receiving a given packet reply with an explicit acknowledgment within a given time. While the use of acknowledgment requests is optional, every Babel speaker MUST be able to reply to such a request.

An acknowledgment MUST be sent to a unicast destination. On the other hand, acknowledgment requests may be sent to either unicast or multicast destinations, in which case they request an acknowledgment from all of the receiving nodes.

When to request acknowledgments is a matter of local policy; the simplest strategy is to never request acknowledgments and to rely on periodic updates to ensure that any reachable routes are eventually

propagated throughout the routing domain. In order to improve convergence speed and reduce the amount of control traffic, acknowledgment requests MAY be used in order to reliably send urgent updates (Section 3.7.2) and retractions (Section 3.5.5), especially when the number of neighbours on a given interface is small. Since Babel is designed to deal gracefully with packet loss on unreliable media, sending all packets with acknowledgment requests is not necessary, and NOT RECOMMENDED, as the acknowledgments cause additional traffic and may force additional Address Resolution Protocol (ARP) or Neighbour Discovery (ND) exchanges.

3.4. Neighbour Acquisition

Neighbour acquisition is the process by which a Babel node discovers the set of neighbours heard over each of its interfaces and ascertains bidirectional reachability. On unreliable media, neighbour acquisition additionally provides some statistics that may be useful for link quality computation.

Before it can exchange routing information with a neighbour, a Babel node MUST create an entry for that neighbour in the neighbour table. When to do that is implementation-specific; suitable strategies include creating an entry when any Babel packet is received, or creating an entry when a Hello TLV is parsed. Similarly, in order to conserve system resources, an implementation SHOULD discard an entry when it has been unused for long enough; suitable strategies include dropping the neighbour after a timeout, and dropping a neighbour when the associated Hello histories become empty (see Appendix A.2).

3.4.1. Reverse Reachability Detection

Every Babel node sends Hello TLVs to its neighbours to indicate that it is alive, at regular or irregular intervals. Each Hello TLV carries an increasing (modulo 2^16) sequence number and an upper bound on the time interval until the next Hello of the same type (see below). If the time interval is set to 0, then the Hello TLV does not establish a new promise: the deadline carried by the previous Hello of the same type still applies to the next Hello (if the most recent scheduled Hello of the right kind was received at time t0 and carried interval i, then the previous promise of sending another Hello before time t0 + i still holds). We say that a Hello is "scheduled" if it carries a non-zero interval, and "unscheduled" otherwise.

There are two kinds of Hellos: Multicast Hellos, which use a per-interface Hello counter (the Multicast Hello seqno), and Unicast Hellos, which use a per-neighbour counter (the Multicast Hello seqno). A Multicast Hello with a given seqno MUST be sent to all

neighbours on a given interface, either by sending it to a multicast address or by sending it to one unicast address per neighbour (hence, the term "Multicast Hello" is a slight misnomer). A Unicast Hello carrying a given seqno should normally be sent to just one neighbour (over unicast), since the sequence numbers of different neighbours are not in general synchronised.

Multicast Hellos sent over multicast can be used for neighbour discovery; hence, a node SHOULD send periodic (scheduled) Multicast Hellos unless neighbour discovery is performed by means outside of the Babel protocol. A node MAY send Unicast Hellos or unscheduled Hellos of either kind for any reason, such as reducing the amount of multicast traffic or improving reliability on link technologies with poor support for link-layer multicast.

A node MAY send a scheduled Hello ahead of time. A node MAY change its scheduled Hello interval. The Hello interval MAY be decreased at any time; it MAY be increased immediately before sending a Hello TLV, but SHOULD NOT be increased at other times. (Equivalently, a node SHOULD send a scheduled Hello immediately after increasing its Hello interval.)

How to deal with received Hello TLVs and what statistics to maintain are considered local implementation matters; typically, a node will maintain some sort of history of recently received Hellos. An example of a suitable algorithm is described in Appendix A.1.

After receiving a Hello, or determining that it has missed one, the node recomputes the association's cost (Section 3.4.3) and runs the route selection procedure (Section 3.6).

3.4.2. Bidirectional Reachability Detection

In order to establish bidirectional reachability, every node sends periodic IHU ("I Heard You") TLVs to each of its neighbours. Since IHUs carry an explicit interval value, they MAY be sent less often than Hellos in order to reduce the amount of routing traffic in dense networks; in particular, they SHOULD be sent less often than Hellos over links with little packet loss. While IHUs are conceptually unicast, they MAY be sent to a multicast address in order to avoid an ARP or Neighbour Discovery exchange and to aggregate multiple IHUs into a single packet.

In addition to the periodic IHUs, a node MAY, at any time, send an unscheduled IHU packet. It MAY also, at any time, decrease its IHU interval, and it MAY increase its IHU interval immediately before sending an IHU, but SHOULD NOT increase it at any other time.

(Equivalently, a node SHOULD send an extra IHU immediately after increasing its Hello interval.)

Every IHU TLV contains two pieces of data: the link's rxcost (reception cost) from the sender's perspective, used by the neighbour for computing link costs (Section 3.4.3), and the interval between periodic IHU packets. A node receiving an IHU sets the value of the txcost (transmission cost) maintained in the neighbour table to the value contained in the IHU, and resets the IHU timer associated to this neighbour to a small multiple of the interval value received in the IHU. When a neighbour's IHU timer expires, the neighbour's txcost is set to infinity.

After updating a neighbour's txcost, the receiving node recomputes the neighbour's cost (Section 3.4.3) and runs the route selection procedure (Section 3.6).

3.4.3. Cost Computation

A neighbourship association's link cost is computed from the values maintained in the neighbour table: the statistics kept in the neighbour table about the reception of Hellos, and the txcost computed from received IHU packets.

For every neighbour, a Babel node computes a value known as this neighbour's rxcost. This value is usually derived from the Hello history, which may be combined with other data, such as statistics maintained by the link layer. The rxcost is sent to a neighbour in each IHU.

Since nodes do not necessarily send periodic Unicast Hellos but do usually send periodic Multicast Hellos (Section 3.4.1), a node SHOULD use an algorithm that yields a finite rxcost when only Multicast Hellos are received, unless interoperability with nodes that only send Multicast Hellos is not required.

How the txcost and rxcost are combined in order to compute a link's cost is a matter of local policy; as far as Babel's correctness is concerned, only the following conditions MUST be satisfied:

- the cost is strictly positive;
- if no Hello TLVs of either kind were received recently, then the cost is infinite;
- if the txcost is infinite, then the cost is infinite.

Note that while this document does not constrain cost computation any further, not all cost computation strategies will give good results. See Appendix A.2 for examples of strategies for computing a link's cost that are known to work well in practice.

3.5. Routing Table Maintenance

Conceptually, a Babel update is a quintuple (prefix, plen, router-id, seqno, metric), where (prefix, plen) is the prefix for which a route is being advertised, router-id is the router-id of the router originating this update, seqno is a nondecreasing (modulo 2^16) integer that carries the originating router seqno, and metric is the announced metric.

Before being accepted, an update is checked against the feasibility condition (Section 3.5.1), which ensures that the route does not create a routing loop. If the feasibility condition is not satisfied, the update is either ignored or prevents the route from being selected, as described in Section 3.5.4. If the feasibility condition is satisfied, then the update cannot possibly cause a routing loop.

3.5.1. The Feasibility Condition

The feasibility condition is applied to all received updates. The feasibility condition compares the metric in the received update with the metrics of the updates previously sent by the receiving node; updates that fail the feasibility condition, and therefore have metrics large enough to cause a routing loop, are either ignored or prevent the resulting route from being selected.

A feasibility distance is a pair (seqno, metric), where seqno is an integer modulo 2^16 and metric is a positive integer. Feasibility distances are compared lexicographically, with the first component inverted: we say that a distance (seqno, metric) is strictly better than a distance (seqno', metric'), written

(seqno, metric) < (seqno', metric')

when

seqno > seqno' or (seqno = seqno' and metric < metric')

where sequence numbers are compared modulo 2^16.

Given a source (prefix, plen, router-id), a node's feasibility distance for this source is the minimum, according to the ordering defined above, of the distances of all the finite updates ever sent

by this particular node for the prefix (prefix, plen) and the given router-id. Feasibility distances are maintained in the source table, the exact procedure is given in Section 3.7.3.

A received update is feasible when either it is a retraction (its metric is FFFF hexadecimal), or the advertised distance is strictly better, in the sense defined above, than the feasibility distance for the corresponding source. More precisely, a route advertisement carrying the quintuple (prefix, plen, router-id, seqno, metric) is feasible if one of the following conditions holds:

- metric is infinite; or
- no entry exists in the source table indexed by (prefix, plen, router-id); or
- an entry (prefix, plen, router-id, seqno', metric') exists in the source table, and either
  - seqno' < seqno or
  - seqno = seqno' and metric < metric'.

Note that the feasibility condition considers the metric advertised by the neighbour, not the route's metric; hence, a fluctuation in a neighbour's cost cannot render a selected route unfeasible. Note further that retractions (updates with infinite metric) are always feasible, since they cannot possibly cause a routing loop.

3.5.2. Metric Computation

A route's metric is computed from the metric advertised by the neighbour and the neighbour's link cost. Just like cost computation, metric computation is considered a local policy matter; as far as Babel is concerned, the function M(c, m) used for computing a metric from a locally computed link cost and the metric advertised by a neighbour MUST only satisfy the following conditions:

- if c is infinite, then M(c, m) is infinite;
- M is strictly monotonic: M(c, m) > m.

Additionally, the metric SHOULD satisfy the following condition:

- M is left-distributive: if m <= m', then M(c, m) <= M(c, m').

Note that while strict monotonicity is essential to the integrity of the network (persistent routing loops may arise if it is not

satisfied), left distributivity is not: if it is not satisfied, Babel will still converge to a loop-free configuration, but might not reach a global optimum (in fact, a global optimum may not even exist).

As with cost computation, not all strategies for computing route metrics will give good results. In particular, some metrics are more likely than others to lead to routing instabilities (route flapping). In Appendix A.3, we give a number of examples of strictly monotonic, left-distributive routing metrics that are known to work well in practice.

3.5.3. Encoding of Updates

In a large network, the bulk of Babel traffic consists of route updates; hence, some care has been given to encoding them efficiently. An Update TLV itself only contains the prefix, seqno, and metric, while the next hop is derived either from the network-layer source address of the packet or from an explicit Next Hop TLV in the same packet. The router-id is derived from a separate Router-Id TLV in the same packet, which optimises the case when multiple updates are sent with the same router-id.

Additionally, a prefix of the advertised prefix can be omitted in an Update TLV, in which case it is copied from a previous Update TLV in the same packet -- this is known as address compression (Section 4.6.9).

Finally, as a special optimisation for the case when a router-id coincides with the interface-id part of an IPv6 address, the router-id can optionally be derived from the low-order bits of the advertised prefix.

The encoding of updates is described in detail in Section 4.6.

3.5.4. Route Acquisition

When a Babel node receives an update (prefix, plen, router-id, seqno, metric) from a neighbour neigh with a link cost value equal to cost, it checks whether it already has a route table entry indexed by (prefix, plen, neigh).

If no such entry exists:

- if the update is unfeasible, it MAY be ignored;
- if the metric is infinite (the update is a retraction of a route we do not know about), the update is ignored;

- otherwise, a new entry is created in the route table, indexed by (prefix, plen, neigh), with source equal to (prefix, plen, router-id), seqno equal to seqno and an advertised metric equal to the metric carried by the update.

If such an entry exists:

- if the entry is currently selected, the update is unfeasible, and the router-id of the update is equal to the router-id of the entry, then the update MAY be ignored;

- otherwise, the entry's sequence number, advertised metric, metric, and router-id are updated and, if the advertised metric is not infinite, the route's expiry timer is reset to a small multiple of the Interval value included in the update. If the update is unfeasible, then the (now unfeasible) entry MUST be immediately unselected. If the update caused the router-id of the entry to change, an update (possibly a retraction) MUST be sent in a timely manner (see Section 3.7.2).

Note that the route table may contain unfeasible routes, either because they were created by an unfeasible update or due to a metric fluctuation. Such routes are never selected, since they are not known to be loop-free; should all the feasible routes become unusable, however, the unfeasible routes can be made feasible and therefore possible to select by sending requests along them (see Section 3.8.2).

When a route's expiry timer triggers, the behaviour depends on whether the route's metric is finite. If the metric is finite, it is set to infinity and the expiry timer is reset. If the metric is already infinite, the route is flushed from the route table.

After the route table is updated, the route selection procedure (Section 3.6) is run.

3.5.5. Hold Time

When a prefix P is retracted, because all routes are unfeasible or have an infinite metric (whether due to the expiry timer or to other reasons), and a shorter prefix P' that covers P is reachable, P' cannot in general be used for routing packets destined to P without running the risk of creating a routing loop (Section 2.8).

To avoid this issue, whenever a prefix P is retracted, a route table entry with infinite metric is maintained as described in Section 3.5.4 above. As long as this entry is maintained, packets destined to an address within P MUST NOT be forwarded by following a

route for a shorter prefix. This entry is removed as soon as a finite-metric update for prefix P is received and the resulting route selected. If no such update is forthcoming, the infinite metric entry SHOULD be maintained at least until it is guaranteed that no neighbour has selected the current node as next-hop for prefix P. This can be achieved by either:

- waiting until the route's expiry timer has expired (Section 3.5.4);
- sending a retraction with an acknowledgment request (Section 3.3) to every reachable neighbour that has not explicitly retracted prefix P and waiting for all acknowledgments.

The former option is simpler and ensures that at that point, any routes for prefix P pointing at the current node have expired. However, since the expiry time can be as high as a few minutes, doing that prevents automatic aggregation by creating spurious black-holes for aggregated routes. The latter option is RECOMMENDED as it dramatically reduces the time for which a prefix is unreachable in the presence of aggregated routes.

3.6. Route Selection

Route selection is the process by which a single route for a given prefix is selected to be used for forwarding packets and to be re-advertised to a node's neighbours.

Babel is designed to allow flexible route selection policies. As far as the protocol's correctness is concerned, the route selection policy MUST only satisfy the following properties:

- a route with infinite metric (a retracted route) is never selected;
- an unfeasible route is never selected.

Note, however, that Babel does not naturally guarantee the stability of routing, and configuring conflicting route selection policies on different routers may lead to persistent route oscillation.

Route selection is a difficult problem, since a good route selection policy needs to take into account multiple mutually contradictory criteria; in roughly decreasing order of importance, these are:

- routes with a small metric should be preferred to routes with a large metric;

- switching router-ids should be avoided;
- routes through stable neighbours should be preferred to routes through unstable ones;
- stable routes should be preferred to unstable ones;
- switching next hops should be avoided.

A simple but useful strategy is to choose the feasible route with the smallest metric, with a small amount of hysteresis applied to avoid switching router-ids too often.

After the route selection procedure is run, triggered updates (Section 3.7.2) and requests (Section 3.8.2) are sent.

3.7. Sending Updates

A Babel speaker advertises to its neighbours its set of selected routes. Normally, this is done by sending one or more multicast packets containing Update TLVs on all of its connected interfaces; however, on link technologies where multicast is significantly more expensive than unicast, a node MAY choose to send multiple copies of updates in unicast packets, especially when the number of neighbours is small.

Additionally, in order to ensure that any black-holes are reliably cleared in a timely manner, a Babel node sends retractions (updates with an infinite metric) for any recently retracted prefixes.

If an update is for a route injected into the Babel domain by the local node (e.g., it carries the address of a local interface, the prefix of a directly attached network, or a prefix redistributed from a different routing protocol), the router-id is set to the local node's router-id, the metric is set to some arbitrary finite value (typically 0), and the seqno is set to the local router's sequence number.

If an update is for a route learned from another Babel speaker, the router-id and sequence number are copied from the route table entry, and the metric is computed as specified in Section 3.5.2.

3.7.1. Periodic Updates

Every Babel speaker periodically advertises all of its selected routes on all of its interfaces, including any recently retracted routes. Since Babel doesn't suffer from routing loops (there is no

"counting to infinity") and relies heavily on triggered updates (Section 3.7.2), this full dump only needs to happen infrequently.

3.7.2. Triggered Updates

In addition to periodic routing updates, a Babel speaker sends unscheduled, or triggered, updates in order to inform its neighbours of a significant change in the network topology.

A change of router-id for the selected route to a given prefix may be indicative of a routing loop in formation; hence, a node MUST send a triggered update in a timely manner whenever it changes the selected router-id for a given destination. Additionally, it SHOULD make a reasonable attempt at ensuring that all reachable neighbours receive this update.

There are two strategies for ensuring that. If the number of neighbours is small, then it is reasonable to send the update together with an acknowledgment request; the update is resent until all neighbours have acknowledged the packet, up to some number of times. If the number of neighbours is large, however, requesting acknowledgments from all of them might cause a non-negligible amount of network traffic; in that case, it may be preferable to simply repeat the update some reasonable number of times (say, 5 for wireless and 2 for wired links).

A route retraction is somewhat less worrying: if the route retraction doesn't reach all neighbours, a black-hole might be created, which, unlike a routing loop, does not endanger the integrity of the network. When a route is retracted, a node SHOULD send a triggered update and SHOULD make a reasonable attempt at ensuring that all neighbours receive this retraction.

Finally, a node MAY send a triggered update when the metric for a given prefix changes in a significant manner, due to a received update, because a link's cost has changed, or because a different next hop has been selected. A node SHOULD NOT send triggered updates for other reasons, such as when there is a minor fluctuation in a route's metric, when the selected next hop changes, or to propagate a new sequence number (except to satisfy a request, as specified in Section 3.8).

3.7.3. Maintaining Feasibility Distances

Before sending an update (prefix, plen, router-id, seqno, metric) with finite metric (i.e., not a route retraction), a Babel node updates the feasibility distance maintained in the source table. This is done as follows.

If no entry indexed by (prefix, plen, router-id) exists in the source table, then one is created with value (prefix, plen, router-id, seqno, metric).

If an entry (prefix, plen, router-id, seqno', metric') exists, then it is updated as follows:

- if seqno > seqno', then seqno' := seqno, metric' := metric;
- if seqno = seqno' and metric' > metric, then metric' := metric;
- otherwise, nothing needs to be done.

The garbage-collection timer for the entry is then reset. Note that the feasibility distance is not updated and the garbage-collection timer is not reset when a retraction (an update with infinite metric) is sent.

When the garbage-collection timer expires, the entry is removed from the source table.

3.7.4. Split Horizon

When running over a transitive, symmetric link technology, e.g., a point-to-point link or a wired LAN technology such as Ethernet, a Babel node SHOULD use an optimisation known as split horizon. When split horizon is used on a given interface, a routing update for prefix P is not sent on the particular interface over which the selected route towards prefix P was learnt.

Split horizon SHOULD NOT be applied to an interface unless the interface is known to be symmetric and transitive; in particular, split horizon is not applicable to decentralised wireless link technologies (e.g., IEEE 802.11 in ad hoc mode) when routing updates are sent over multicast.

3.8. Explicit Requests

In normal operation, a node's route table is populated by the regular and triggered updates sent by its neighbours. Under some circumstances, however, a node sends explicit requests in order to cause a resynchronisation with the source after a mobility event or to prevent a route from spuriously expiring.

The Babel protocol provides two kinds of explicit requests: route requests, which simply request an update for a given prefix, and seqno requests, which request an update for a given prefix with a

specific sequence number. The former are never forwarded; the latter are forwarded if they cannot be satisfied by the receiver.

3.8.1. Handling Requests

Upon receiving a request, a node either forwards the request or sends an update in reply to the request, as described in the following sections. If this causes an update to be sent, the update is either sent to a multicast address on the interface on which the request was received, or to the unicast address of the neighbour that sent the request.

The exact behaviour is different for route requests and seqno requests.

3.8.1.1. Route Requests

When a node receives a route request for a given prefix, it checks its route table for a selected route to this exact prefix. If such a route exists, it MUST send an update (over unicast or over multicast); if such a route does not exist, it MUST send a retraction for that prefix.

When a node receives a wildcard route request, it SHOULD send a full route table dump. Full route dumps MAY be rate-limited, especially if they are sent over multicast.

3.8.1.2. Seqno Requests

When a node receives a seqno request for a given router-id and sequence number, it checks whether its route table contains a selected entry for that prefix. If a selected route for the given prefix exists, it has finite metric, and either the router-ids are different or the router-ids are equal and the entry's sequence number is no smaller (modulo 2^16) than the requested sequence number, the node MUST send an update for the given prefix. If the router-ids match but the requested seqno is larger (modulo 2^16) than the route entry's, the node compares the router-id against its own router-id. If the router-id is its own, then it increases its sequence number by 1 (modulo 2^16) and sends an update. A node MUST NOT increase its sequence number by more than 1 in response to a seqno request.

Otherwise, if the requested router-id is not its own, the received request's hop count is 2 or more, and the node is advertising the prefix to its neighbours, the node selects a neighbour to forward the request to as follows:

- o if the node has one or more feasible routes toward the requested prefix with a next hop that is not the requesting node, then the node MUST forward the request to the next hop of one such route;

- o otherwise, if the node has one or more (not necessarily feasible) routes to the requested prefix with a next hop that is not the requesting node, then the node SHOULD forward the request to the next hop of one such route.

In order to actually forward the request, the node decrements the hop count and sends the request in a unicast packet destined to the selected neighbour.

A node SHOULD maintain a list of recently forwarded seqno requests and forward the reply (an update with a seqno sufficiently large to satisfy the request) in a timely manner. A node SHOULD compare every incoming seqno request against its list of recently forwarded seqno requests and avoid forwarding it if it is redundant (i.e., if it has recently sent a request with the same prefix, router-id and a seqno that is not smaller modulo 2^16).

Since the request-forwarding mechanism does not necessarily obey the feasibility condition, it may get caught in routing loops; hence, requests carry a hop count to limit the time during which they remain in the network. However, since requests are only ever forwarded as unicast packets, the initial hop count need not be kept particularly low, and performing an expanding horizon search is not necessary. A single request MUST NOT be duplicated: it MUST NOT be forwarded to a multicast address, and it MUST NOT be forwarded to multiple neighbours. However, if a seqno request is resent by its originator, the subsequent copies MAY be forwarded to a different neighbour than the initial one.

3.8.2. Sending Requests

A Babel node MAY send a route or seqno request at any time, to a multicast or a unicast address; there is only one case when originating requests is required (Section 3.8.2.1).

3.8.2.1. Avoiding Starvation

When a route is retracted or expires, a Babel node usually switches to another feasible route for the same prefix. It may be the case, however, that no such routes are available.

A node that has lost all feasible routes to a given destination but still has unexpired unfeasible routes to that destination MUST send a seqno request; if it doesn’t have any such routes, it MAY still send

a seqno request. The router-id of the request is set to the router-id of the route that it has just lost, and the requested seqno is the value contained in the source table plus 1.

If the node has any (unfeasible) routes to the requested destination, then it MUST send the request to at least one of the next-hop neighbours that advertised these routes, and SHOULD send it to all of them; in any case, it MAY send the request to any other neighbours, whether they advertise a route to the requested destination or not. A simple implementation strategy is therefore to unconditionally multicast the request over all interfaces.

Similar requests will be sent by other nodes that are affected by the route's loss. If the network is still connected, and assuming no packet loss, then at least one of these requests will be forwarded to the source, resulting in a route being advertised with a new sequence number. (Due to duplicate suppression, only a small number of such requests will actually reach the source.)

In order to compensate for packet loss, a node SHOULD repeat such a request a small number of times if no route becomes feasible within a short time. In the presence of heavy packet loss, however, all such requests might be lost; in that case, the mechanism in the next section will eventually ensure that a new seqno is received.

3.8.2.2. Dealing with Unfeasible Updates

When a route's metric increases, a node might receive an unfeasible update for a route that it has currently selected. As specified in Section 3.5.1, the receiving node will either ignore the update or unselect the route.

In order to keep routes from spuriously expiring because they have become unfeasible, a node SHOULD send a unicast seqno request when it receives an unfeasible update for a route that is currently selected. The requested sequence number is computed from the source table as in Section 3.8.2.1 above.

Additionally, since metric computation does not necessarily coincide with the delay in propagating updates, a node might receive an unfeasible update from a currently unselected neighbour that is preferable to the currently selected route (e.g., because it has a much smaller metric); in that case, the node SHOULD send a unicast seqno request to the neighbour that advertised the preferable update.

3.8.2.3. Preventing Routes from Expiring

In normal operation, a route's expiry timer never triggers: since a route's hold time is computed from an explicit interval included in Update TLVs, a new update (possibly a retraction) should arrive in time to prevent a route from expiring.

In the presence of packet loss, however, it may be the case that no update is successfully received for an extended period of time, causing a route to expire. In order to avoid such spurious expiry, shortly before a selected route expires, a Babel node SHOULD send a unicast route request to the neighbour that advertised this route; since nodes always send either updates or retractions in response to non-wildcard route requests (Section 3.8.1.1), this will usually result in the route being either refreshed or retracted.

3.8.2.4. Acquiring New Neighbours

In order to speed up convergence after a mobility event, a node MAY send a unicast wildcard request after acquiring a new neighbour. Additionally, a node MAY send a small number of multicast wildcard requests shortly after booting. Note however that doing that carelessly can cause serious congestion when a whole network is rebooted, especially on link layers with high per-packet overhead (e.g., IEEE 802.11).

4. Protocol Encoding

A Babel packet is sent as the body of a UDP datagram, with network-layer hop count set to 1, destined to a well-known multicast address or to a unicast address, over IPv4 or IPv6; in the case of IPv6, these addresses are link-local. Both the source and destination UDP port are set to a well-known port number. A Babel packet MUST be silently ignored unless its source address is either a link-local IPv6 address or an IPv4 address belonging to the local network, and its source port is the well-known Babel port. It MAY be silently ignored if its destination address is a global IPv6 address.

In order to minimise the number of packets being sent while avoiding lower-layer fragmentation, a Babel node SHOULD attempt to maximise the size of the packets it sends, up to the outgoing interface's MTU adjusted for lower-layer headers (28 octets for UDP over IPv4, 48 octets for UDP over IPv6). It MUST NOT send packets larger than the attached interface's MTU adjusted for lower-layer headers or 512 octets, whichever is larger, but not exceeding 2^16 - 1 adjusted for lower-layer headers. Every Babel speaker MUST be able to receive packets that are as large as any attached interface's MTU adjusted

for lower-layer headers or 512 octets, whichever is larger. Babel packets MUST NOT be sent in IPv6 Jumbograms.

In order to avoid global synchronisation of a Babel network and to aggregate multiple TLVs into large packets, a Babel node SHOULD buffer every TLV and delay sending a packet by a small, randomly chosen delay [JITTER]. In order to allow accurate computation of packet loss rates, this delay MUST NOT be larger than half the advertised Hello interval.

4.1. Data Types

4.1.1. Interval

Relative times are carried as 16-bit values specifying a number of centiseconds (hundredths of a second). This allows times up to roughly 11 minutes with a granularity of 10ms, which should cover all reasonable applications of Babel.

4.1.2. Router-Id

A router-id is an arbitrary 8-octet value. A router-id MUST NOT consist of either all zeroes or all ones.

4.1.3. Address

Since the bulk of the protocol is taken by addresses, multiple ways of encoding addresses are defined. Additionally, a common subnet prefix may be omitted when multiple addresses are sent in a single packet -- this is known as address compression (Section 4.6.9).

Address encodings:

- AE 0: wildcard address. The value is 0 octets long.
- AE 1: IPv4 address. Compression is allowed. 4 octets or less.
- AE 2: IPv6 address. Compression is allowed. 16 octets or less.
- AE 3: link-local IPv6 address. Compression is not allowed. The value is 8 octets long, a prefix of fe80::/64 is implied.

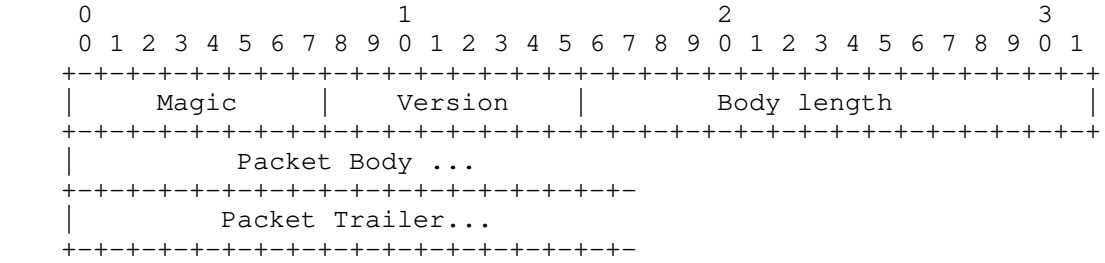
The address family associated to an address encoding is either IPv4 or IPv6; it is undefined for AE 0, IPv4 for AE 1, and IPv6 for AEs 2 and 3.

4.1.4. Prefixes

A network prefix is encoded just like a network address, but it is stored in the smallest number of octets that are enough to hold the significant bits (up to the prefix length).

4.2. Packet Format

A Babel packet consists of a 4-octet header, followed by a sequence of TLVs (the packet body), optionally followed by a second sequence of TLVs (the packet trailer).



Fields :

Magic The arbitrary but carefully chosen value 42 (decimal); packets with a first octet different from 42 MUST be silently ignored.

Version This document specifies version 2 of the Babel protocol. Packets with a second octet different from 2 MUST be silently ignored.

Body length The length in octets of the body following the packet header (excluding the Magic, Version and Body length fields, and excluding the packet trailer).

Packet Body The packet body; a sequence of TLVs.

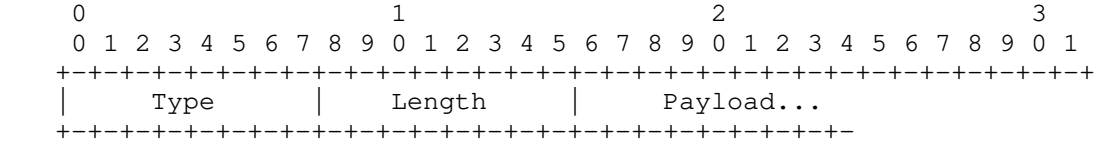
Packet Trailer The packet trailer; another sequence of TLVs.

The packet body and the packet trailer are both sequences of TLVs. A TLV may appear in the packet trailer only if it is explicitly allowed to do so: the receiver MUST silently ignore any TLV that appears in the packet trailer unless it is explicitly specified to be allowed in the packet trailer. Among the TLVs defined in this document, only Pad1 and PadN are allowed in the packet trailer; since these TLVs are ignored in any case, an implementation MAY silently ignore the packet

trailer unless it implements at least one extension that uses the packet trailer.

4.3. TLV Format

With the exception of Pad1, all TLVs have the following structure:



Fields :

Type The type of the TLV.

Length The length of the body, exclusive of the Type and Length fields. If the body is longer than the expected length of a given type of TLV, any extra data MUST be silently ignored.

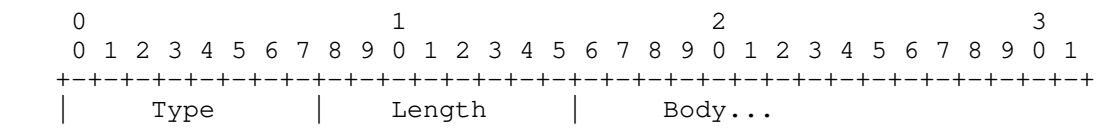
Payload The TLV payload, which consists of a body and, for selected TLV types, an optional list of sub-TLVs.

TLVs with an unknown type value MUST be silently ignored.

4.4. Sub-TLV Format

Every TLV carries an explicit length in its header; however, most TLVs are self-terminating, in the sense that it is possible to determine the length of the body without reference to the explicit Length field. If a TLV has a self-terminating format, then it MAY allow a sequence of sub-TLVs to follow the body.

Sub-TLVs have the same structure as TLVs. With the exception of PAD1, all TLVs have the following structure:



Fields :

Type The type of the sub-TLV.

Length The length of the body, in octets, exclusive of the Type and Length fields.

Body The sub-TLV body, the interpretation of which depends on both the type of the sub-TLV and the type of the TLV within which it is embedded.

The most-significant bit of the sub-TLV, called the mandatory bit, indicates how to handle unknown sub-TLVs. If the mandatory bit is not set, then an unknown sub-TLV MUST be silently ignored, and the rest of the TLV processed normally. If the mandatory bit is set, then the whole enclosing TLV MUST be silently ignored (except for updating the parser state by a Router-Id, Next-Hop or Update TLV, see Section 4.6.7, Section 4.6.8, and Section 4.6.9).

4.5. Parser state

Babel uses a stateful parser: a TLV may refer to data from a previous TLV. The parser state consists of the following pieces of data:

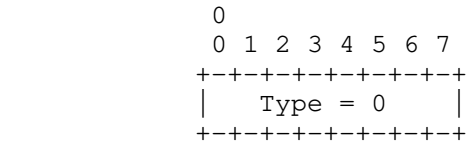
- for each address encoding that allows compression, the current default prefix; this is undefined at the start of the packet, and is updated by each Update TLV with the Prefix flag set (Section 4.6.9);
- for each address family (IPv4 or IPv6), the current next-hop; this is the source address of the enclosing packet for the matching address family at the start of a packet, and is updated by each Next-Hop TLV (Section 4.6.8);
- the current router-id; this is undefined at the start of the packet, and is updated by each Router-ID TLV (Section 4.6.7) and by each Update TLV with Router-Id flag set.

Since the parser state is separate from the bulk of Babel's state, and since for correct parsing it must be identical across implementations, it is updated before checking for mandatory TLVs: parsing a TLV MUST update the parser state even if the TLV is otherwise ignored due to an unknown mandatory sub-TLV.

None of the TLVs that modify the parser state are allowed in the packet trailer; hence, an implementation may choose to use a dedicated stateless parser to parse the packet trailer.

4.6. Details of Specific TLVs

4.6.1. Pad1

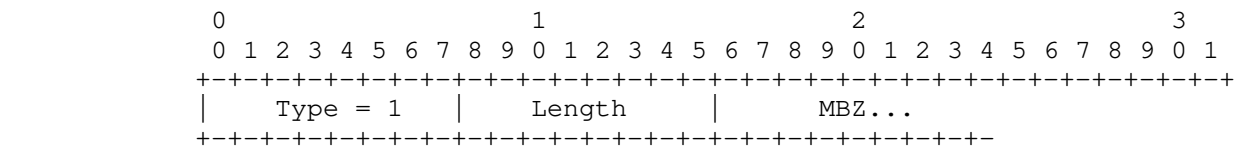


Fields :

Type Set to 0 to indicate a Pad1 TLV.

This TLV is silently ignored on reception. It is allowed in the packet trailer.

4.6.2. PadN



Fields :

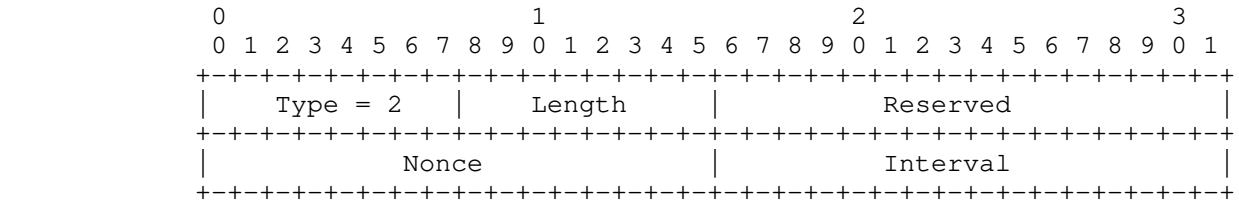
Type Set to 1 to indicate a PadN TLV.

Length The length of the body, exclusive of the Type and Length fields.

MBZ Set to 0 on transmission.

This TLV is silently ignored on reception. It is allowed in the packet trailer.

4.6.3. Acknowledgment Request



This TLV requests that the receiver send an Acknowledgment TLV within the number of centiseconds specified by the Interval field.

Fields :

Type Set to 2 to indicate an Acknowledgment Request TLV.

Length The length of the body, exclusive of the Type and Length fields.

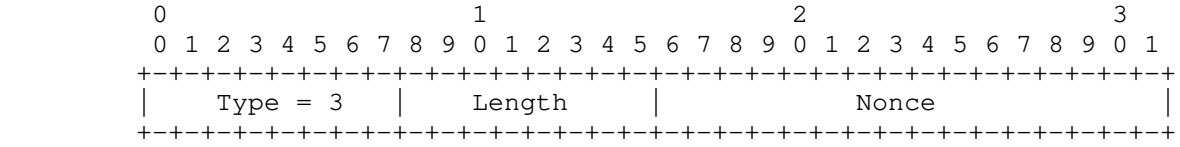
Reserved Sent as 0 and MUST be ignored on reception.

Nonce An arbitrary value that will be echoed in the receiver's Acknowledgment TLV.

Interval A time interval in centiseconds after which the sender will assume that this packet has been lost. This MUST NOT be 0. The receiver MUST send an Acknowledgment TLV before this time has elapsed (with a margin allowing for propagation time).

This TLV is self-terminating, and allows sub-TLVs.

4.6.4. Acknowledgment



This TLV is sent by a node upon receiving an Acknowledgment Request.

Fields :

Type Set to 3 to indicate an Acknowledgment TLV.

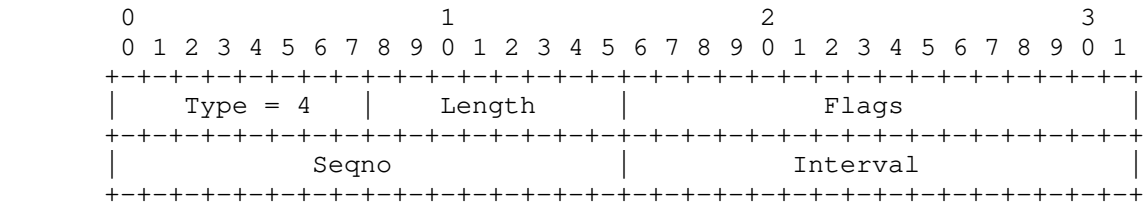
Length The length of the body, exclusive of the Type and Length fields.

Nonce Set to the Nonce value of the Acknowledgment Request that prompted this Acknowledgment.

Since nonce values are not globally unique, this TLV MUST be sent to a unicast address.

This TLV is self-terminating, and allows sub-TLVs.

4.6.5. Hello



This TLV is used for neighbour discovery and for determining a neighbour's reception cost.

Fields :

Type Set to 4 to indicate a Hello TLV.

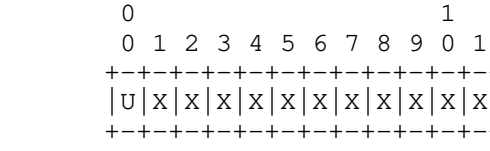
Length The length of the body, exclusive of the Type and Length fields.

Flags The individual bits of this field specify special handling of this TLV (see below).

Seqno If the Unicast flag is set, this is the value of the sending node's outgoing Unicast Hello seqno for this neighbour. Otherwise, it is the sending node's outgoing Multicast Hello seqno for this interface.

Interval If non-zero, this is an upper bound, expressed in centiseconds, on the time after which the sending node will send a new scheduled Hello TLV with the same setting of the Unicast flag. If this is 0, then this Hello represents an unscheduled Hello, and doesn't carry any new information about times at which Hellos are sent.

The Flags field is interpreted as follows:



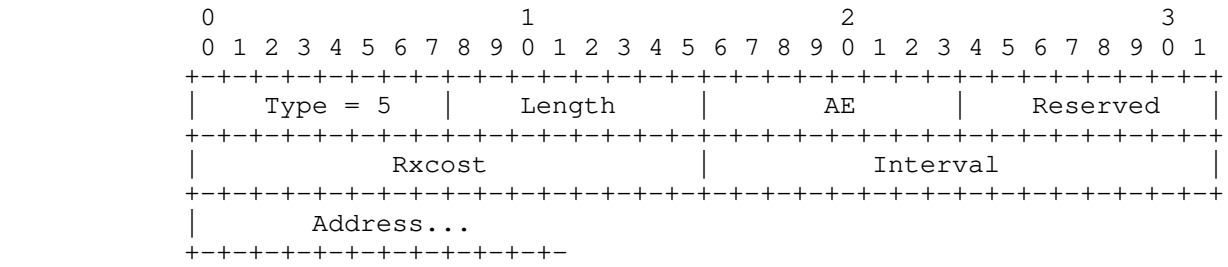
- U (Unicast) flag (8000 hexadecimal): if set, then this Hello represents a Unicast Hello, otherwise it represents a Multicast Hello;

- o X: all other bits MUST be sent as 0 and silently ignored on reception.

Every time a Hello is sent, the corresponding seqno counter MUST be incremented. Since there is a single seqno counter for all the Multicast Hellos sent by a given node over a given interface, if the Unicast flag is not set, this TLV MUST be sent to all neighbors on this link, which can be achieved by sending to a multicast destination, or by sending multiple packets to the unicast addresses of all reachable neighbours. Conversely, if the Unicast flag is set, this TLV MUST be sent to a single neighbour, which can achieved by sending to a unicast destination. In order to avoid large discontinuities in link quality, multiple Hello TLVs SHOULD NOT be sent in the same packet.

This TLV is self-terminating, and allows sub-TLVs.

4.6.6. IHU



An IHU ("I Heard You") TLV is used for confirming bidirectional reachability and carrying a link's transmission cost.

Fields :

Type Set to 5 to indicate an IHU TLV.

Length The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Address field. This should be 1 or 3 in most cases. As an optimisation, it MAY be 0 if the TLV is sent to a unicast address, if the association is over a point-to-point link, or when bidirectional reachability is ascertained by means outside of the Babel protocol.

Reserved Sent as 0 and MUST be ignored on reception.

Rxcost The rxcost according to the sending node of the interface whose address is specified in the Address field. The value FFFF hexadecimal (infinity) indicates that this interface is unreachable.

Interval An upper bound, expressed in centiseconds, on the time after which the sending node will send a new IHU; this MUST NOT be 0. The receiving node will use this value in order to compute a hold time for this symmetric association.

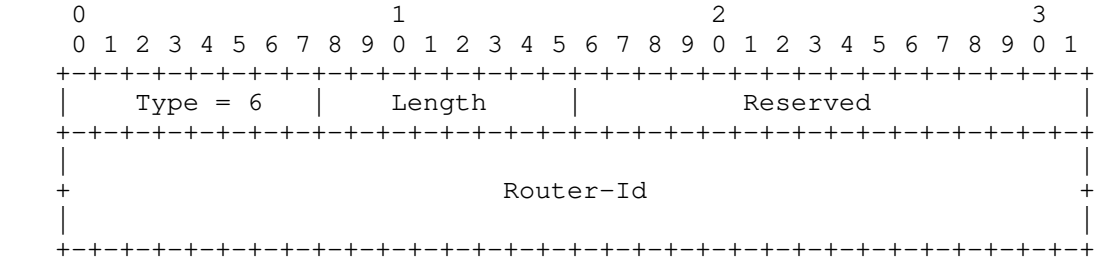
Address The address of the destination node, in the format specified by the AE field. Address compression is not allowed.

Conceptually, an IHU is destined to a single neighbour. However, IHU TLVs contain an explicit destination address, and MAY be sent to a multicast address, as this allows aggregation of IHUs destined to distinct neighbours into a single packet and avoids the need for an ARP or Neighbour Discovery exchange when a neighbour is not being used for data traffic.

IHU TLVs with an unknown value in the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.7. Router-Id



A Router-Id TLV establishes a router-id that is implied by subsequent Update TLVs. This TLV sets the router-id even if it is otherwise ignored due to an unknown mandatory sub-TLV.

Fields :

Type Set to 6 to indicate a Router-Id TLV.

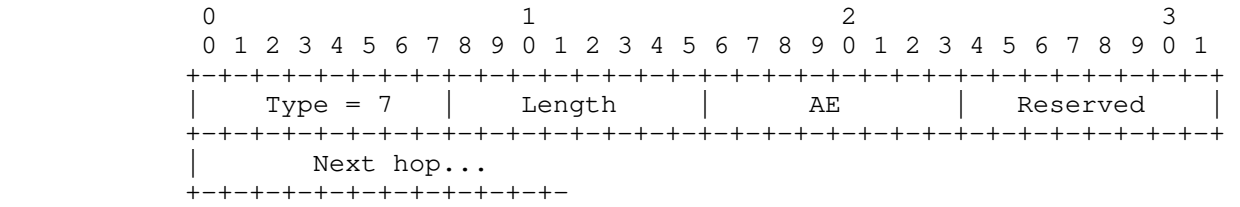
Length The length of the body, exclusive of the Type and Length fields.

Reserved Sent as 0 and MUST be ignored on reception.

Router-Id The router-id for routes advertised in subsequent Update TLVs. This MUST NOT consist of all zeroes or all ones.

This TLV is self-terminating, and allows sub-TLVs.

4.6.8. Next Hop



A Next Hop TLV establishes a next-hop address for a given address family (IPv4 or IPv6) that is implied in subsequent Update TLVs. This TLV sets up the next-hop for subsequent Update TLVs even if it is otherwise ignored due to an unknown mandatory sub-TLV.

Fields :

Type Set to 7 to indicate a Next Hop TLV.

Length The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Address field. This SHOULD be 1 (IPv4) or 3 (link-local IPv6), and MUST NOT be 0.

Reserved Sent as 0 and MUST be ignored on reception.

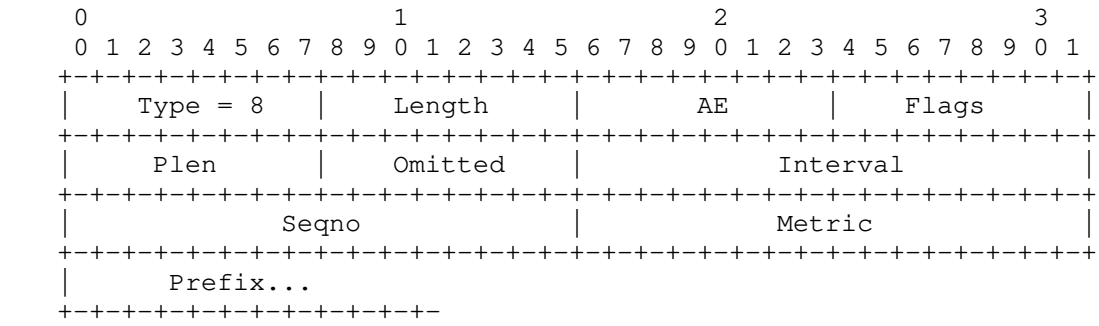
Next hop The next-hop address advertised by subsequent Update TLVs, for this address family.

When the address family matches the network-layer protocol that this packet is transported over, a Next Hop TLV is not needed: in the absence of a Next Hop TLV in a given address family, the next hop address is taken to be the source address of the packet.

Next Hop TLVs with an unknown value for the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.9. Update



An Update TLV advertises or retracts a route. As an optimisation, it can optionally have the side effect of establishing a new implied router-id and a new default prefix.

Fields :

Type Set to 8 to indicate an Update TLV.

Length The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Prefix field.

Flags The individual bits of this field specify special handling of this TLV (see below).

Plen The length of the advertised prefix.

Omitted The number of octets that have been omitted at the beginning of the advertised prefix and that should be taken from a preceding Update TLV in the same address family with the Prefix flag set.

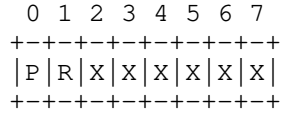
Interval An upper bound, expressed in centiseconds, on the time after which the sending node will send a new update for this prefix. This MUST NOT be 0. The receiving node will use this value to compute a hold time for the route table entry. The value FFFF hexadecimal (infinity) expresses that this announcement will not be repeated unless a request is received (Section 3.8.2.3).

Seqno The originator's sequence number for this update.

Metric The sender's metric for this route. The value FFFF hexadecimal (infinity) means that this is a route retraction.

Prefix The prefix being advertised. This field's size is (Plen/8 - Omitted) rounded upwards.

The Flags field is interpreted as follows:



- P (Prefix) flag (80 hexadecimal): if set, then this Update establishes a new default prefix for subsequent Update TLVs with a matching address encoding within the same packet, even if this TLV is otherwise ignored due to an unknown mandatory sub-TLV;

- R (Router-Id) flag (40 hexadecimal): if set, then this TLV establishes a new default router-id for this TLV and subsequent Update TLVs in the same packet, even if this TLV is otherwise ignored due to an unknown mandatory sub-TLV. This router-id is computed from the first address of the advertised prefix as follows:

  * if the length of the address is 8 octets or more, then the new router-id is taken from the 8 last octets of the address;

  * if the length of the address is smaller than 8 octets, then the new router-id consists of the required number of zero octets followed by the address, i.e., the address is stored on the right of the router-id. For example, for an IPv4 address, the router-id consists of 4 octets of zeroes followed by the IPv4 address.

- X: all other bits MUST be sent as 0 and silently ignored on reception.

The prefix being advertised by an Update TLV is computed as follows:

- the first Omitted octets of the prefix are taken from the previous Update TLV with the Prefix flag set and the same address encoding, even if it was ignored due to an unknown mandatory sub-TLV;

- the next (Plen/8 - Omitted) rounded upwards octets are taken from the Prefix field;

o the remaining octets are set to 0. If AE is 3 (link-local IPv6), Omitted MUST be 0)

If the Metric field is finite, the router-id of the originating node for this announcement is taken from the prefix advertised by this Update if the Router-Id flag is set, computed as described above. Otherwise, it is taken either from the preceding Router-Id packet, or the preceding Update packet with the Router-Id flag set, whichever comes last, even if that TLV is otherwise ignored due to an unknown mandatory sub-TLV.

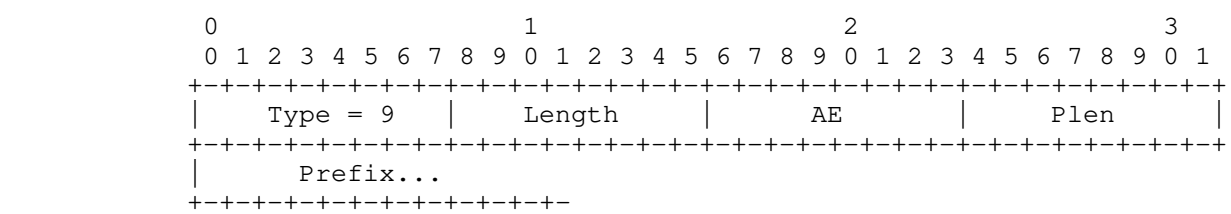
The next-hop address for this update is taken from the last preceding Next Hop TLV with a matching address family (IPv4 or IPv6) in the same packet even if it was otherwise ignored due to an unknown mandatory sub-TLV; if no such TLV exists, it is taken from the network-layer source address of this packet.

If the metric field is FFFF hexadecimal, this TLV specifies a retraction. In that case, the router-id, next-hop and seqno are not used. AE MAY then be 0, in which case this Update retracts all of the routes previously advertised by the sending interface. If the metric is finite, AE MUST NOT be 0. If the metric is infinite and AE is 0, Plen and Omitted MUST both be 0.

Update TLVs with an unknown value in the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.10. Route Request



A Route Request TLV prompts the receiver to send an update for a given prefix, or a full route table dump.

Fields :

Type Set to 9 to indicate a Route Request TLV.

Length The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Prefix field. The value 0 specifies that this is a request for a full route table dump (a wildcard request).

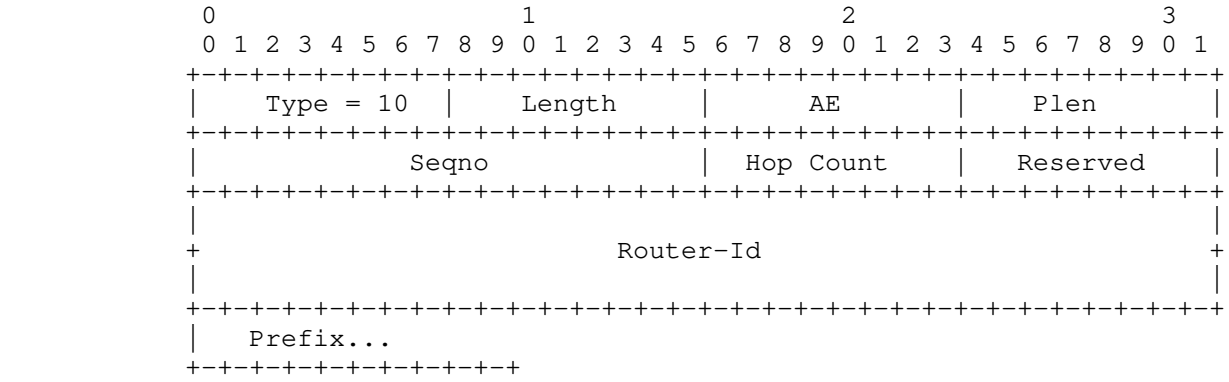
Plen The length of the requested prefix.

Prefix The prefix being requested. This field's size is Plen/8 rounded upwards.

A Request TLV prompts the receiver to send an update message (possibly a retraction) for the prefix specified by the AE, Plen, and Prefix fields, or a full dump of its route table if AE is 0 (in which case Plen MUST be 0 and Prefix is of length 0).

This TLV is self-terminating, and allows sub-TLVs.

4.6.11. Seqno Request



A Seqno Request TLV prompts the receiver to send an Update for a given prefix with a given sequence number, or to forward the request further if it cannot be satisfied locally.

Fields :

Type Set to 10 to indicate a Seqno Request message.

Length The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Prefix field. This MUST NOT be 0.

Plen The length of the requested prefix.

Seqno The sequence number that is being requested.

Hop Count The maximum number of times that this TLV may be forwarded, plus 1. This MUST NOT be 0.

Reserved Sent as 0 and MUST be ignored on reception.

Router Id The Router-Id that is being requested. This MUST NOT consist of all zeroes or all ones.

Prefix The prefix being requested. This field's size is Plen/8 rounded upwards.

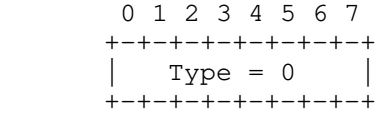
A Seqno Request TLV prompts the receiving node to send a finite-metric Update for the prefix specified by the AE, Plen, and Prefix fields, with either a router-id different from what is specified by the Router-Id field, or a Seqno no less (modulo 2^16) than what is specified by the Seqno field. If this request cannot be satisfied locally, then it is forwarded according to the rules set out in Section 3.8.1.2.

While a Seqno Request MAY be sent to a multicast address, it MUST NOT be forwarded to a multicast address and MUST NOT be forwarded to more than one neighbour. A request MUST NOT be forwarded if its Hop Count field is 1.

This TLV is self-terminating, and allows sub-TLVs.

4.7. Details of specific sub-TLVs

4.7.1. Pad1

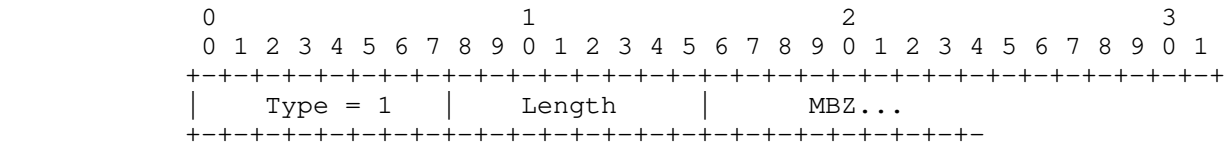


Fields :

Type Set to 0 to indicate a Pad1 sub-TLV.

This sub-TLV is silently ignored on reception. It is allowed within any TLV that allows sub-TLVs.

4.7.2. PadN



Fields :

Type Set to 1 to indicate a PadN sub-TLV.

Length The length of the body, in octets, exclusive of the Type and Length fields.

MBZ Set to 0 on transmission.

This sub-TLV is silently ignored on reception. It is allowed within any TLV that allows sub-TLVs.

5. IANA Considerations

IANA has registered the UDP port number 6696, called "babel", for use by the Babel protocol.

IANA has registered the IPv6 multicast group ff02::1:6 and the IPv4 multicast group 224.0.0.111 for use by the Babel protocol.

IANA has created a registry called "Babel TLV Types". The values in this registry are not changed by this specification.

IANA has created a registry called "Babel sub-TLV Types". Due to the addition of a Mandatory bit to the Babel protocol, the values in the "Babel sub-TLV Types" registry are amended as follows:

Type	Name	Reference
0	Pad1	this document
1	PadN	this document
112-126	Reserved for Experimental Use	this document
127	Reserved for expansion of the type space	this document
240-254	Reserved for Experimental Use	this document
255	Reserved for expansion of the type space	this document

Existing assignments in the "Babel sub-TLV Types" registry in the range 2 to 111 are not changed by this specification. The values 224 through 239, previously reserved for Experimental Use, are now unassigned.

IANA has created a registry called "Babel Flags Values". IANA is instructed to rename this registry to "Babel Update Flags Values", with its contents unchanged.

IANA is instructed to create a new registry called "Babel Hello Flags Values". The allocation policy for this registry is Specification Required [RFC5226]. The initial values in this registry are as follows:

Bit	Name	Reference
0	Unicast	this document
1-15	Unassigned	

IANA is instructed to replace all references to RFCs 6126 and 7557 in all of the registries mentioned above by references to this document.

6. Security Considerations

As defined in this document, Babel is a completely insecure protocol. Any attacker can misdirect data traffic by advertising routes with a low metric or a high seqno. This issue can be solved either by a

lower-layer security mechanism (e.g., link-layer security), or by deploying a suitable authentication mechanism within Babel itself. With the exception of Hello TLVs used for discovery, Babel control traffic can be carried over unicast, which makes it possible to protect Babel traffic with a protocol that can only protect unicast data, for example IPsec with IKEv2, or DTLS.

The information that a Babel node announces to the whole routing domain is often sufficient to determine a mobile node's physical location with reasonable precision. The privacy issues that this causes can be mitigated somewhat by using randomly chosen router-ids and randomly chosen IP addresses, and changing them periodically.

When carried over IPv6, Babel packets are ignored unless they are sent from a link-local IPv6 address; since routers don't forward link-local IPv6 packets, this provides protection against spoofed Babel packets being sent from the global Internet. No such natural protection exists when Babel packets are carried over IPv4.

7. Acknowledgments

A number of people have contributed text and ideas to this specification. The authors are particularly indebted to Matthieu Boutier, Gwendoline Chouasne, Margaret Cullen, Donald Eastlake and Toke Hoiland-Jorgensen. Earlier versions of this document greatly benefited from the input of Joel Halpern. The address compression technique was inspired by [PACKETBB].

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017.

8.2. Informative References

[DSDV] Perkins, C. and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers", ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications 234-244, 1994.

[DUAL] Garcia Luna Aceves, J., "Loop-Free Routing Using Diffusing Computations", IEEE/ACM Transactions on Networking 1:1, February 1993.

[EIGRP] Albrightson, B., Garcia Luna Aceves, J., and J. Boyle, "EIGRP -- a Fast Routing Protocol Based on Distance Vectors", Proc. Interop 94, 1994.

[ETX] De Couto, D., Aguayo, D., Bicket, J., and R. Morris, "A high-throughput path metric for multi-hop wireless networks", Proc. MobiCom 2003, 2003.

[IS-IS] "Information technology -- Telecommunications and information exchange between systems -- Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)", ISO/IEC 10589:2002, 2002.

[JITTER] Floyd, S. and V. Jacobson, "The synchronization of periodic routing messages", IEEE/ACM Transactions on Networking 2, 2, 122-136, April 1994.

[OSPF] Moy, J., "OSPF Version 2", RFC 2328, April 1998.

[PACKETBB] Clausen, T., Dearlove, C., Dean, J., and C. Adjih, "Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format", RFC 5444, February 2009.

[RIP] Malkin, G., "RIP Version 2", RFC 2453, November 1998.

Appendix A. Cost and Metric Computation

The strategy for computing link costs and route metrics is a local matter; Babel itself only requires that it comply with the conditions given in Section 3.4.3 and Section 3.5.2. Different nodes may use different strategies in a single network and may use different strategies on different interface types. This section describes the strategies used by the sample implementation of Babel.

The sample implementation of Babel sends periodic Multicast Hellos, and never sends Unicast Hellos. It maintains statistics about the last 16 received Hello TLVs of each kind (Appendix A.1), computes costs by using the 2-out-of-3 strategy (Appendix A.2.1) on wired links, and ETX (Appendix A.2.2) on wireless links. It uses an additive algebra for metric computation (Appendix A.3.1).

A.1. Maintaining Hello History

For each neighbour, the sample implementation of Babel maintains two sets of Hello history, one for each kind of Hello, and an expected sequence number, one for Multicast and one for Unicast Hellos. Each Hello history is a vector of 16 bits, where a 1 value represents a received Hello, and a 0 value a missed Hello. For each kind of Hello, the expected sequence number, written ne, is the sequence number that is expected to be carried by the next received Hello from this neighbour.

Whenever it receives a Hello packet of a given kind from a neighbour, a node compares the received sequence number nr for that kind of Hello with its expected sequence number ne. Depending on the outcome of this comparison, one of the following actions is taken:

- if the two differ by more than 16 (modulo 2^16), then the sending node has probably rebooted and lost its sequence number; the whole associated neighbour table entry is flushed and a new one is created;

- otherwise, if the received nr is smaller (modulo 2^16) than the expected sequence number ne, then the sending node has increased its Hello interval without us noticing; the receiving node removes the last (ne - nr) entries from this neighbour's Hello history (we "undo history");

- otherwise, if nr is larger (modulo 2^16) than ne, then the sending node has decreased its Hello interval, and some Hellos were lost; the receiving node adds (nr - ne) 0 bits to the Hello history (we "fast-forward").

The receiving node then appends a 1 bit to the Hello history and sets ne to (nr + 1). If the Interval field of the received Hello is not zero, it resets the neighbour's hello timer to 1.5 times the advertised Interval (the extra margin allows for delay due to jitter).

Whenever either Hello timer associated to a neighbour expires, the local node adds a 0 bit to this neighbour's Hello history, and increments the expected Hello number. If both Hello histories are

empty (they contain 0 bits only), the neighbour entry is flushed; otherwise, the relevant hello timer is reset to the value advertised in the last Hello of that kind received from this neighbour (no extra margin is necessary in this case, since jitter was already taken into account when computing the timeout that has just expired).

A.2. Cost Computation

This section discusses how to compute costs based on Hello history.

A.2.1. k-out-of-j

K-out-of-j link sensing is suitable for wired links that are either up, in which case they only occasionally drop a packet, or down, in which case they drop all packets.

The k-out-of-j strategy is parameterised by two small integers k and j, such that 0 < k <= j, and the nominal link cost, a constant K >= 1. A node keeps a history of the last j hellos; if k or more of those have been correctly received, the link is assumed to be up, and the rxcost is set to K; otherwise, the link is assumed to be down, and the rxcost is set to infinity.

Since Babel supports two kinds of Hellos, a Babel node performs k-out-of-j twice for each neighbour, once on the Unicast and once on the Multicast Hello history. If either of the instances of k-out-of-j indicates that the link is up, then the link is assumed to be up, and the rxcost is set to K; if both instances indicate that the link is down, then the link is assumed to be down, and the rxcost is set to infinity. In other words, the resulting rxcost is the minimum of the rxcosts yielded by the two instances of k-out-of-j link sensing.

The cost of a link performing k-out-of-j link sensing is defined as follows:

- cost = FFFF hexadecimal if rxcost = FFFF hexadecimal;
- cost = txcost otherwise.

A.2.2. ETX

Unlike wired links, which are bimodal (either up or down), wireless links exhibit continuous variation of the link quality. Naive application of hop-count routing in networks that use wireless links for transit tends to select long, lossy links in preference to shorter, lossless links, which can dramatically reduce throughput.

For that reason, a routing protocol designed to support wireless links must perform some form of link-quality estimation.

ETX [ETX] is a simple link-quality estimation algorithm that is designed to work well with the IEEE 802.11 MAC. By default, the IEEE 802.11 MAC performs ARQ and rate adaptation on unicast frames, but not on multicast frames, which are sent at a fixed rate with no ARQ; therefore, measuring the loss rate of multicast frames yields a useful estimate of a link's quality.

A node performing ETX link quality estimation uses a neighbour's Multicast Hello history to compute an estimate, written beta, of the probability that a Hello TLV is successfully received. Beta can be computed as the fraction of 1 bits within a small number (say, 6) of the most recent entries in the Multicast Hello history, or it can be an exponential average, or some combination of both approaches.

Let alpha be MIN(1, 256/txcost), an estimate of the probability of successfully sending a Hello TLV. The cost is then computed by

```
cost = 256/(alpha * beta)
```

or, equivalently,

```
cost = (MAX(txcost, 256) * rxcost) / 256.
```

Since the IEEE 802.11 MAC performs ARQ on unicast frames, unicast frames do not provide a useful measure of link quality, and therefore ETX ignores the Unicast Hello history. Thus, a node performing ETX link-quality estimation will not route through neighbouring nodes unless they send periodic Multicast Hellos (possibly in addition to Unicast Hellos).

A.3. Metric Computation

As described in Section 3.5.2, the metric advertised by a neighbour is combined with the link cost to yield a metric.

A.3.1. Additive Metrics

The simplest approach for obtaining a monotonic, left-distributive metric is to define the metric of a route as the sum of the costs of the component links. More formally, if a neighbour advertises a route with metric m over a link with cost c, then the resulting route has metric M(c, m) = c + m.

A multiplicative metric can be converted into an additive one by taking the logarithm (in some suitable base) of the link costs.

A.3.2. External Sources of Willingness

A node may want to vary its willingness to forward packets by taking into account information that is external to the Babel protocol, such as the monetary cost of a link, the node's battery status, CPU load, etc. This can be done by adding to every route's metric a value k that depends on the external data. For example, if a battery-powered node receives an update with metric m over a link with cost c, it might compute a metric M(c, m) = k + c + m, where k depends on the battery status.

In order to preserve strict monotonicity (Section 3.5.2), the value k must be greater than -c.

Appendix B. Constants

The choice of time constants is a trade-off between fast detection of mobility events and protocol overhead. Two implementations of Babel with different time constants will interoperate, although the resulting convergence time will most likely be dictated by the slower of the two.

Experience with the sample implementation of Babel indicates that the Hello interval is the most important time constant: a mobility event is detected within 1.5 to 3 Hello intervals. Due to Babel's reliance on triggered updates and explicit requests, the Update interval only has an effect on the time it takes for accurate metrics to be propagated after variations in link costs too small to trigger an unscheduled update or in the presence of packet loss.

At the time of writing, the sample implementation of Babel uses the following default values:

Multicast Hello Interval: 4 seconds.

IHU Interval: the advertised IHU interval is always 3 times the Multicast Hello interval. IHUs are actually sent with each Hello on lossy links (as determined from the Hello history), but only with every third Multicast Hello on lossless links.

Unicast Hello Interval: the sample implementation never sends scheduled Unicast Hellos;

Update Interval: 4 times the Multicast Hello interval.

IHU Hold Time: 3.5 times the advertised IHU interval.

Route Expiry Time: 3.5 times the advertised update interval.

Source GC time: 3 minutes.

Request timeout: initially 2 seconds, doubled every time a request is resent, up to a maximum of three times.

The amount of jitter applied to a packet depends on whether it contains any urgent TLVs or not (Section 3.1). Urgent triggered updates and urgent requests are delayed by no more than 200ms; acknowledgments, by no more than the associated deadline; and other TLVs by no more than one-half the Multicast Hello interval.

Appendix C. Considerations for protocol extensions

Babel is an extensible protocol, and this document defines a number of mechanisms that can be used to extend the protocol in a backwards compatible manner:

- increasing the version number in the packet header;
- defining new TLVs;
- defining new sub-TLVs (with or without the mandatory bit set);
- defining new AEs;
- using the packet trailer.

This appendix is intended to guide designers of protocol extensions in chosing a particular encoding.

The version number in the Babel header should only be increased if the new version is not backwards compatible with the original protocol.

In many cases, an extension could be implemented either by defining a new TLV, or by adding a new sub-TLV to an existing TLV. For example, an extension whose purpose is to attach additional data to route updates can be implemented either by creating a new "enriched" Update TLV, by adding a non-mandatory sub-TLV to the Update TLV, or by adding a mandatory sub-TLV.

The various encodings are treated differently by implementations that do not understand the extension. In the case of a new TLV or of a sub-TLV with the mandatory bit set, the whole TLV is ignored by implementations that do not implement the extension, while in the case of a non-mandatory sub-TLV, the TLV is parsed and acted upon, and only the unknown sub-TLV is silently ignored. Therefore, a non-mandatory sub-TLV should be used by extensions that extend the Update

in a compatible manner (the extension data may be silently ignored), while a mandatory sub-TLV or a new TLV must be used by extensions that make incompatible extensions to the meaning of the TLV (the whole TLV must be thrown away if the extension data is not understood).

Experience shows that the need for additional data tends to crop up in the most unexpected places. Hence, it is recommended that extensions that define new TLVs should make them self-terminating, and allow attaching sub-TLVs to them.

Adding a new AE is essentially equivalent to adding a new TLV: Update TLVs with an unknown AE are ignored, just like unknown TLVs. However, adding a new AE is more involved than adding a new TLV, since it creates a new set of compression state. Additionally, since the Next Hop TLV creates state specific to a given address family, as opposed to a given AE, a new AE for a previously defined address family must not be used in the Next Hop TLV if backwards compatibility is required. A similar issue arises with Update TLVs with unknown AEs establishing a new router-id (due to the Router-Id flag being set). Therefore, defining new AEs must be done with care if compatibility with unextended implementations is required.

The packet trailer is intended to carry cryptographic signatures that only cover the packet body; storing the cryptographic signatures in the packet trailer avoids clearing the signature before computing a hash of the packet body, and makes it possible to check a cryptographic signature before running the full, stateful TLV parser. Thus, any TLV that is allowed to appear in the packet trailer must not need to be protected by cryptographic security protocols, it should be easy to parse, and should not require stateful parsing.

Appendix D. Stub Implementations

Babel is a fairly economic protocol. Updates take between 12 and 40 octets per destination, depending on the address family and how successful compression is; in a double-stack flat network, an average of less than 24 octets per update is typical. The route table occupies about 35 octets per IPv6 entry. To put these values into perspective, a single full-size Ethernet frame can carry some 65 route updates, and a megabyte of memory can contain a 20000-entry route table and the associated source table.

Babel is also a reasonably simple protocol. The sample implementation consists of less than 12 000 lines of C code, and it compiles to less than 120 kB of text on a 32-bit CISC architecture; about half of this figure is due to protocol extensions and user-interface code.

Nonetheless, in some very constrained environments, such as PDAs, microwave ovens, or abacuses, it may be desirable to have subset implementations of the protocol.

There are many different definitions of a stub router, but for the needs of this section a stub implementation of Babel is one that announces one or more directly attached prefixes into a Babel network but doesn't reannounce any routes that it has learnt from its neighbours. It may either maintain a full routing table, or simply select a default gateway amongst any one of its neighbours that announces a default route. Since a stub implementation never forwards packets except from or to directly attached links, it cannot possibly participate in a routing loop, and hence it need not evaluate the feasibility condition or maintain a source table.

No matter how primitive, a stub implementation MUST parse sub-TLVs attached to any TLVs that it understands and check the mandatory bit. It MUST answer acknowledgment requests and MUST participate in the Hello/IHU protocol. It MUST also be able to reply to seqno requests for routes that it announces and SHOULD be able to reply to route requests.

Experience shows that an IPv6-only stub implementation of Babel can be written in less than 1000 lines of C code and compile to 13 kB of text on 32-bit CISC architecture.

Appendix E. Software Availability

The sample implementation of Babel is available from <https://www.irif.fr/~jch/software/babel/>.

Appendix F. Changes from previous versions

F.1. Changes since RFC 6126

- Changed UDP port number to 6696.
- Consistently use router-id rather than id.
- Clarified that the source garbage collection timer is reset after sending an update even if the entry was not modified.
- In section "Seqno Requests", fixed an erroneous "route request".
- In the description of the Seqno Request TLV, added the description of the Router-Id field.
- Made router-ids all-0 and all-1 forbidden.

F.2. Changes since draft-ietf-babel-rfc6126bis-00

- Added security considerations.

F.3. Changes since draft-ietf-babel-rfc6126bis-01

- Integrated the format of sub-TLVs.
- Mentioned for each TLV whether it supports sub-TLVs.
- Added Appendix C.
- Added a mandatory bit in sub-TLVs.
- Changed compression state to be per-AF rather than per-AE.
- Added implementation hint for the routing table.
- Clarified how router-ids are computed when bit 0x40 is set in Updates.
- Relaxed the conditions for sending requests, and tightened the conditions for forwarding requests.
- Clarified that neighbours should be acquired at some point, but it doesn't matter when.

F.4. Changes since draft-ietf-babel-rfc6126bis-02

- Added Unicast Hellos.
- Added unscheduled (interval-less) Hellos.
- Changed Appendix A to consider Unicast and unscheduled Hellos.
- Changed Appendix B to agree with the reference implementation.
- Added optional algorithm to avoid the hold time.
- Changed the table of pending seqno requests to be indexed by router-id in addition to prefixes.
- Relaxed the route acquisition algorithm.
- Replaced minimal implementations by stub implementations.
- Added acknowledgments section.

F.5. Changes since draft-ietf-babel-rfc6126bis-03

- Clarified that all the data structures are conceptual.
- Made sending and receiving Multicast Hellos a SHOULD, avoids expressing any opinion about Unicast Hellos.
- Removed opinion about Multicast vs. Unicast Hellos (Appendix A.4).
- Made hold-time into a SHOULD rather than MUST.
- Clarified that Seqno Requests are for a finite-metric Update.
- Clarified that sub-TLVs Pad1 and PadN are allowed within any TLV that allows sub-TLVs.
- Updated IANA Considerations.
- Updated Security Considerations.
- Renamed routing table back to route table.
- Made buffering outgoing updates a SHOULD.
- Weakened advice to use modified EUI-64 in router-ids.
- Added information about sending requests to Appendix B.
- A number of minor wording changes and clarifications.

F.6. Changes since draft-ietf-babel-rfc6126bis-03

Minor editorial changes.

F.7. Changes since draft-ietf-babel-rfc6126bis-04

- Renamed isotonicity to left-distributivity.
- Minor clarifications to unicast hellos.
- Updated requirements boilerplate to RFC 8174.
- Minor editorial changes.

F.8. Changes since draft-ietf-babel-rfc6126bis-05

- Added information about the packet trailer, now that it is used by draft-ietf-babel-hmac.

Authors' Addresses

Juliusz Chroboczek
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: jch@irif.fr

David Schinazi
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
US

Email: dschinazi@apple.com

Network Working Group
Internet-Draft
Obsoletes: 6126,7557 (if approved)
Intended status: Standards Track
Expires: February 26, 2021

J. Chroboczek
IRIF, University of Paris-Diderot
D. Schinazi
Google LLC

August 25, 2020

The Babel Routing Protocol
draft-ietf-babel-rfc6126bis-20

Abstract

Babel is a loop-avoiding distance-vector routing protocol that is robust and efficient both in ordinary wired networks and in wireless mesh networks. This document describes the Babel routing protocol, and obsoletes RFCs 6126 and 7557.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on February 26, 2021.

Copyright Notice

Table of Contents

1. Introduction

Babel is a loop-avoiding distance-vector routing protocol that is designed to be robust and efficient both in networks using prefix-based routing and in networks using flat routing ("mesh networks"), and both in relatively stable wired networks and in highly dynamic wireless networks. This document describes the Babel routing protocol, and obsoletes [RFC6126] and [RFC7557].

1.1. Features

The main property that makes Babel suitable for unstable networks is that, unlike naive distance-vector routing protocols [RIP], it strongly limits the frequency and duration of routing pathologies such as routing loops and black-holes during reconvergence. Even after a mobility event is detected, a Babel network usually remains loop-free. Babel then quickly reconverges to a configuration that preserves the loop-freedom and connectedness of the network, but is not necessarily optimal; in many cases, this operation requires no packet exchanges at all. Babel then slowly converges, in a time on the scale of minutes, to an optimal configuration. This is achieved by using sequenced routes, a technique pioneered by Destination-Sequenced Distance-Vector routing [DSDV].

More precisely, Babel has the following properties:

- when every prefix is originated by at most one router, Babel never suffers from routing loops;
- when a single prefix is originated by multiple routers, Babel may occasionally create a transient routing loop for this particular prefix; this loop disappears in time proportional to the loop's diameter, and never again (up to an arbitrary garbage-collection (GC) time) will the routers involved participate in a routing loop for the same prefix;
- assuming bounded packet loss rates, any routing black-holes that may appear after a mobility event are corrected in a time at most proportional to the network's diameter.

Babel has provisions for link quality estimation and for fairly arbitrary metrics. When configured suitably, Babel can implement shortest-path routing, or it may use a metric based, for example, on measured packet loss.

Babel nodes will successfully establish an association even when they are configured with different parameters. For example, a mobile node that is low on battery may choose to use larger time constants (hello and update intervals, etc.) than a node that has access to wall power. Conversely, a node that detects high levels of mobility may choose to use smaller time constants. The ability to build such heterogeneous networks makes Babel particularly adapted to the unmanaged or wireless environment.

Finally, Babel is a hybrid routing protocol, in the sense that it can carry routes for multiple network-layer protocols (IPv4 and IPv6), regardless of which protocol the Babel packets are themselves being carried over.

1.2. Limitations

Babel has two limitations that make it unsuitable for use in some environments. First, Babel relies on periodic routing table updates rather than using a reliable transport; hence, in large, stable networks it generates more traffic than protocols that only send updates when the network topology changes. In such networks, protocols such as OSPF [OSPF], IS-IS [IS-IS], or the Enhanced Interior Gateway Routing Protocol (EIGRP) [EIGRP] might be more suitable.

Second, unless the second algorithm described in Section 3.5.4 is implemented, Babel does impose a hold time when a prefix is

retracted. While this hold time does not apply to the exact prefix being retracted, and hence does not prevent fast reconvergence should it become available again, it does apply to any shorter prefix that covers it. This may make those implementations of Babel that do not implement the optional algorithm described in Section 3.5.4 unsuitable for use in networks that implement automatic prefix aggregation.

1.3. Specification of Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Conceptual Description of the Protocol

Babel is a loop-avoiding distance vector protocol: it is based on the Bellman-Ford algorithm, just like the venerable RIP [RIP], but includes a number of refinements that either prevent loop formation altogether, or ensure that a loop disappears in a timely manner and doesn't form again.

Conceptually, Bellman-Ford is executed in parallel for every source of routing information (destination of data traffic). In the following discussion, we fix a source S; the reader will recall that the same algorithm is executed for all sources.

2.1. Costs, Metrics and Neighbourship

For every pair of neighbouring nodes A and B, Babel computes an abstract value known as the cost of the link from A to B, written C(A, B). Given a route between any two (not necessarily neighbouring) nodes, the metric of the route is the sum of the costs of all the links along the route. The goal of the routing algorithm is to compute, for every source S, the tree of routes of lowest metric to S.

Costs and metrics need not be integers. In general, they can be values in any algebra that satisfies two fairly general conditions (Section 3.5.2).

A Babel node periodically sends Hello messages to all of its neighbours; it also periodically sends an IHU ("I Heard You") message to every neighbour from which it has recently heard a Hello. From the information derived from Hello and IHU messages received from its

neighbour B, a node A computes the cost C(A, B) of the link from A to B.

2.2. The Bellman-Ford Algorithm

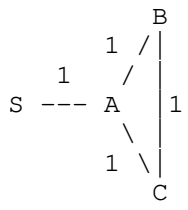
Every node A maintains two pieces of data: its estimated distance to S, written D(A), and its next-hop router to S, written NH(A). Initially, D(S) = 0, D(A) is infinite, and NH(A) is undefined.

Periodically, every node B sends to all of its neighbours a route update, a message containing D(B). When a neighbour A of B receives the route update, it checks whether B is its selected next hop; if that is the case, then NH(A) is set to B, and D(A) is set to C(A, B) + D(B). If that is not the case, then A compares C(A, B) + D(B) to its current value of D(A). If that value is smaller, meaning that the received update advertises a route that is better than the currently selected route, then NH(A) is set to B, and D(A) is set to C(A, B) + D(B).

A number of refinements to this algorithm are possible, and are used by Babel. In particular, convergence speed may be increased by sending unscheduled "triggered updates" whenever a major change in the topology is detected, in addition to the regular, scheduled updates. Additionally, a node may maintain a number of alternate routes, which are being advertised by neighbours other than its selected neighbour, and which can be used immediately if the selected route were to fail.

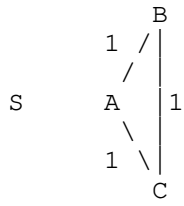
2.3. Transient Loops in Bellman-Ford

It is well known that a naive application of Bellman-Ford to distributed routing can cause transient loops after a topology change. Consider for example the following topology:



After convergence, D(B) = D(C) = 2, with NH(B) = NH(C) = A.

Suppose now that the link between S and A fails:



When it detects the failure of the link, A switches its next hop to B (which is still advertising a route to S with metric 2), and advertises a metric equal to 3, and then advertises a new route with metric 3. This process of nodes changing selected neighbours and increasing their metric continues until the advertised metric reaches "infinity", a value larger than all the metrics that the routing protocol is able to carry.

2.4. Feasibility Conditions

Bellman-Ford is a very robust algorithm: its convergence properties are preserved when routers delay route acquisition or when they discard some updates. Babel routers discard received route announcements unless they can prove that accepting them cannot possibly cause a routing loop.

More formally, we define a condition over route announcements, known as the "feasibility condition", that guarantees the absence of routing loops whenever all routers ignore route updates that do not satisfy the feasibility condition. In effect, this makes Bellman-Ford into a family of routing algorithms, parameterised by the feasibility condition.

Many different feasibility conditions are possible. For example, BGP can be modelled as being a distance-vector protocol with a (rather drastic) feasibility condition: a routing update is only accepted when the receiving node's AS number is not included in the update's AS-Path attribute (note that BGP's feasibility condition does not ensure the absence of transient "micro-loops" during reconvergence).

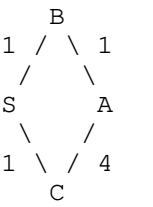
Another simple feasibility condition, used in the Destination-Sequenced Distance-Vector (DSDV) routing protocol [DSDV] and in the Ad hoc On-Demand Distance Vector (AODV) protocol [RFC3561], stems from the following observation: a routing loop can only arise after a router has switched to a route with a larger metric than the route that it had previously selected. Hence, one may define that a route is feasible when its metric at the local node would be no larger than the metric of the currently selected route, i.e., an announcement carrying a metric D(B) is accepted by A when C(A, B) + D(B) <= D(A). If all routers obey this constraint, then the metric at every router

is nonincreasing, and the following invariant is always preserved: if A has selected B as its next hop, then D(B) < D(A), which implies that the forwarding graph is loop-free.

Babel uses a slightly more refined feasibility condition, derived from EIGRP [DUAL]. Given a router A, define the feasibility distance of A, written FD(A), as the smallest metric that A has ever advertised for S to any of its neighbours. An update sent by a neighbour B of A is feasible when the metric D(B) advertised by B is strictly smaller than A's feasibility distance, i.e., when D(B) < FD(A).

It is easy to see that this latter condition is no more restrictive than DSDV-feasibility. Suppose that node A obeys DSDV-feasibility; then D(A) is nonincreasing, hence at all times D(A) <= FD(A). Suppose now that A receives a DSDV-feasible update that advertises a metric D(B). Since the update is DSDV-feasible, C(A, B) + D(B) <= D(A), hence D(B) < D(A), and since D(A) <= FD(A), D(B) < FD(A).

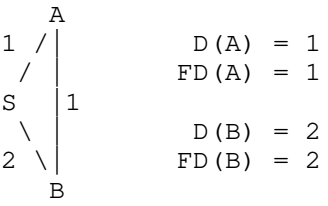
To see that it is strictly less restrictive, consider the following diagram, where A has selected the route through B, and D(A) = FD(A) = 2. Since D(C) = 1 < FD(A), the alternate route through C is feasible for A, although its metric C(A, C) + D(C) = 5 is larger than that of the currently selected route:



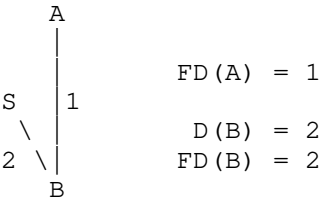
To show that this feasibility condition still guarantees loop-freedom, recall that at the time when A accepts an update from B, the metric D(B) announced by B is no smaller than FD(B); since it is smaller than FD(A), at that point in time FD(B) < FD(A). Since this property is preserved both when A sends updates and when it picks a different next hop, it remains true at all times, which ensures that the forwarding graph has no loops.

2.5. Solving Starvation: Sequencing Routes

Obviously, the feasibility conditions defined above cause starvation when a router runs out of feasible routes. Consider the following diagram, where both A and B have selected the direct route to S:



Suppose now that the link between A and S breaks:

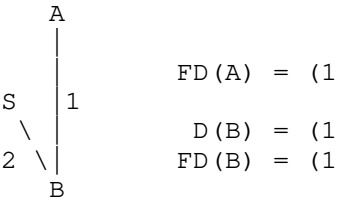


The only route available from A to S, the one that goes through B, is not feasible: A suffers from spurious starvation. At that point, the whole subtree suffering from starvation must be reset, which is essentially what EIGRP does when it performs a global synchronisation of all the routers in the starving subtree (the "active" phase of EIGRP).

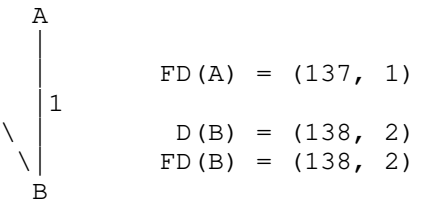
Babel reacts to starvation in a less drastic manner, by using sequenced routes, a technique introduced by DSDV and adopted by AODV. In addition to a metric, every route carries a sequence number, a nondecreasing integer that is propagated unchanged through the network and is only ever incremented by the source; a pair (s, m), where s is a sequence number and m a metric, is called a distance.

A received update is feasible when either it is more recent than the feasibility distance maintained by the receiving node, or it is equally recent and the metric is strictly smaller. More formally, if FD(A) = (s, m), then an update carrying the distance (s', m') is feasible when either s' > s, or s = s' and m' < m.

Assuming the sequence number of S is 137, the diagram above becomes:



After S increases its sequence number, and the new sequence number is propagated to B, we have:



at which point the route through B becomes feasible again.

Note that while sequence numbers are used for determining feasibility, they are not used in route selection: a node ignores the sequence number when selecting the best route to a given destination (Section 3.6). Doing otherwise would cause route oscillation while a sequence number propagates through the network, and might even cause persistent blackholes with some exotic metrics.

2.6. Requests

In DSDV, the sequence number of a source is increased periodically. A route becomes feasible again after the source increases its sequence number, and the new sequence number is propagated through the network, which may, in general, require a significant amount of time.

Babel takes a different approach. When a node detects that it is suffering from a potentially spurious starvation, it sends an explicit request to the source for a new sequence number. This request is forwarded hop by hop to the source, with no regard to the feasibility condition. Upon receiving the request, the source increases its sequence number and broadcasts an update, which is forwarded to the requesting node.

Note that after a change in network topology not all such requests will, in general, reach the source, as some will be sent over links that are now broken. However, if the network is still connected, then at least one among the nodes suffering from spurious starvation has an (unfeasible) route to the source; hence, in the absence of packet loss, at least one such request will reach the source. (Resending requests a small number of times compensates for packet loss.)

Since requests are forwarded with no regard to the feasibility condition, they may, in general, be caught in a forwarding loop; this

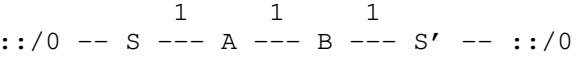
is avoided by having nodes perform duplicate detection for the requests that they forward.

2.7. Multiple Routers

The above discussion assumes that each prefix is originated by a single router. In real networks, however, it is often necessary to have a single prefix originated by multiple routers: for example, the default route will be originated by all of the edge routers of a routing domain.

Since synchronising sequence numbers between distinct routers is problematic, Babel treats routes for the same prefix as distinct entities when they are originated by different routers: every route announcement carries the router-id of its originating router, and feasibility distances are not maintained per prefix, but per source, where a source is a pair of a router-id and a prefix. In effect, Babel guarantees loop-freedom for the forwarding graph to every source; since the union of multiple acyclic graphs is not in general acyclic, Babel does not in general guarantee loop-freedom when a prefix is originated by multiple routers, but any loops will be broken in a time at most proportional to the diameter of the loop -- as soon as an update has "gone around" the routing loop.

Consider for example the following topology, where A has selected the default route through S, and B has selected the one through S':



Suppose that both default routes fail at the same time; then nothing prevents A from switching to B, and B simultaneously switching to A. However, as soon as A has successfully advertised the new route to B, the route through A will become unfeasible for B. Conversely, as soon as B will have advertised the route through A, the route through B will become unfeasible for A.

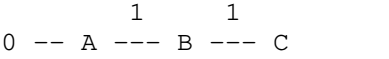
In effect, the routing loop disappears at the latest when routing information has gone around the loop. Since this process can be delayed by lost packets, Babel makes certain efforts to ensure that updates are sent reliably after a router-id change (Section 3.7.2).

Additionally, after the routers have advertised the two routes, both sources will be in their source tables, which will prevent them from ever again participating in a routing loop involving routes from S and S' (up to the source GC time, which, available memory permitting, can be set to arbitrarily large values).

2.8. Overlapping Prefixes

In the above discussion, we have assumed that all prefixes are disjoint, as is the case in flat ("mesh") routing. In practice, however, prefixes may overlap: for example, the default route overlaps with all of the routes present in the network.

After a route fails, it is not correct in general to switch to a route that subsumes the failed route. Consider for example the following configuration:



Suppose that node C fails. If B forwards packets destined to C by following the default route, a routing loop will form, and persist until A learns of B's retraction of the direct route to C. B avoids this pitfall by installing an "unreachable" route after a route is retracted; this route is maintained until it can be guaranteed that the former route has been retracted by all of B's neighbours (Section 3.5.4).

3. Protocol Operation

Every Babel speaker is assigned a router-id, which is an arbitrary string of 8 octets that is assumed unique across the routing domain. For example, router-ids could be assigned randomly, or they could be derived from a link-layer address. (The protocol encoding is slightly more compact when router-ids are assigned in the same manner as the IPv6 layer assigns host IDs; see the definition of the "R" flag in Section 4.6.9.)

3.1. Message Transmission and Reception

Babel protocol packets are sent in the body of a UDP datagram (as described in Section 4 below). Each Babel packet consists of zero or more TLVs. Most TLVs may contain sub-TLVs.

The protocol's control traffic can be carried indifferently over IPv6 or over IPv4, and prefixes of either address family can be announced over either protocol. Thus, there are at least two natural deployment models: using IPv6 exclusively for all control traffic, or running two distinct protocol instances, one for each address family. The exclusive use of IPv6 for all control traffic is RECOMMENDED, since using both protocols at the same time doubles the amount of traffic devoted to neighbour discovery and link quality estimation.

The source address of a Babel packet is always a unicast address, link-local in the case of IPv6. Babel packets may be sent to a well-known (link-local) multicast address or to a (link-local) unicast address. In normal operation, a Babel speaker sends both multicast and unicast packets to its neighbours.

With the exception of acknowledgments, all Babel TLVs can be sent to either unicast or multicast addresses, and their semantics does not depend on whether the destination is a unicast or a multicast address. Hence, a Babel speaker does not need to determine the destination address of a packet that it receives in order to interpret it.

A moderate amount of jitter may be applied to packets sent by a Babel speaker: outgoing TLVs are buffered and SHOULD be sent with a random delay. This is done for two purposes: it avoids synchronisation of multiple Babel speakers across a network [JITTER], and it allows for the aggregation of multiple TLVs into a single packet.

The maximum amount of delay a TLV can be subjected to depends on the TLV. When the protocol description specifies that a TLV is urgent (as in Section 3.7.2 and Section 3.8.2), then the TLV MUST be sent within a short time known as the urgent timeout (see Appendix B for recommended values). When the TLV is governed by a timeout explicitly included in a previous TLV, such as in the case of Acknowledgements (Section 4.6.4), Updates (Section 3.7) and IHUs (Section 3.4.2), then the TLV MUST be sent early enough to meet the explicit deadline (with a small margin to allow for propagation delays). In all other cases, the TLV SHOULD be sent out within one-half of the Multicast Hello interval.

In order to avoid packet drops (either at the sender or at the receiver), a delay SHOULD be introduced between successive packets sent out on the same interface, within the constraints of the previous paragraph. Note however that such packet pacing might impair the ability of some link layers (e.g., IEEE 802.11 [IEEE802.11]) to perform packet aggregation.

3.2. Data Structures

In this section, we give a description of the data structures that every Babel speaker maintains. This description is conceptual: a Babel speaker may use different data structures as long as the resulting protocol is the same as the one described in this document. For example, rather than maintaining a single table containing both selected and unselected (fallback) routes, as described in Section 3.2.6 below, an actual implementation would probably use two tables, one with selected routes and one with fallback routes.

3.2.1. Sequence number arithmetic

Sequence numbers (seqnos) appear in a number of Babel data structures, and they are interpreted as integers modulo 2^16. For the purposes of this document, arithmetic on sequence numbers is defined as follows.

Given a seqno s and a non-negative integer n, the sum of s and n is defined by

s + n (modulo 2^16) = (s + n) MOD 2^16

or, equivalently,

s + n (modulo 2^16) = (s + n) AND 65535

where MOD is the modulo operation yielding a non-negative integer and AND is the bitwise conjunction operation.

Given two sequence numbers s and s', the relation s is less than s' (s < s') is defined by

s < s' (modulo 2^16) when 0 < ((s' - s) MOD 2^16) < 32768

or equivalently

s < s' (modulo 2^16) when s /= s' and ((s' - s) AND 32768) = 0.

3.2.2. Node Sequence Number

A node's sequence number is a 16-bit integer that is included in route updates sent for routes originated by this node.

A node increments its sequence number (modulo 2^16) whenever it receives a request for a new sequence number (Section 3.8.1.2). A node SHOULD NOT increment its sequence number (seqno) spontaneously, since increasing seqnos makes it less likely that other nodes will have feasible alternate routes when their selected routes fail.

3.2.3. The Interface Table

The interface table contains the list of interfaces on which the node speaks the Babel protocol. Every interface table entry contains the interface's outgoing Multicast Hello seqno, a 16-bit integer that is sent with each Multicast Hello TLV on this interface and is incremented (modulo 2^16) whenever a Multicast Hello is sent. (Note that an interface's Multicast Hello seqno is unrelated to the node's seqno.)

There are two timers associated with each interface table entry. The periodic Multicast Hello timer governs the sending of scheduled Multicast Hello and IHU packets (Section 3.4. The periodic Update timer governs the sending of periodic route updates (Section 3.7.1). See Appendix B for suggested time constants.

3.2.4. The Neighbour Table

The neighbour table contains the list of all neighbouring interfaces from which a Babel packet has been recently received. The neighbour table is indexed by pairs of the form (interface, address), and every neighbour table entry contains the following data:

- the local node's interface over which this neighbour is reachable;
- the address of the neighbouring interface;
- a history of recently received Multicast Hello packets from this neighbour; this can, for example, be a sequence of n bits, for some small value n, indicating which of the n hellos most recently sent by this neighbour have been received by the local node;
- a history of recently received Unicast Hello packets from this neighbour;
- the "transmission cost" value from the last IHU packet received from this neighbour, or FFFF hexadecimal (infinity) if the IHU hold timer for this neighbour has expired;
- the expected incoming Multicast Hello sequence number for this neighbour, an integer modulo 2^16.
- the expected incoming Unicast Hello sequence number for this neighbour, an integer modulo 2^16.
- the outgoing Unicast Hello sequence number for this neighbour, an integer modulo 2^16 that is sent with each Unicast Hello TLV to this neighbour and is incremented (modulo 2^16) whenever a Unicast Hello is sent. (Note that the outgoing Unicast Hello seqno for a neighbour is distinct from the interface's outgoing Multicast Hello seqno.)

There are three timers associated with each neighbour entry -- the multicast hello timer, which is set to the interval value carried by scheduled Multicast Hello TLVs sent by this neighbour, the unicast hello timer, which is set to the interval value carried by scheduled Unicast Hello TLVs, and the IHU timer, which is set to a small

multiple of the interval carried in IHU TLVs (see "IHU Hold time" in Appendix B for suggested values).

Note that the neighbour table is indexed by IP addresses, not by router-ids: neighbourship is a relationship between interfaces, not between nodes. Therefore, two nodes with multiple interfaces can participate in multiple neighbourship relationships, a situation that can notably arise when wireless nodes with multiple radios are involved.

3.2.5. The Source Table

The source table is used to record feasibility distances. It is indexed by triples of the form (prefix, plen, router-id), and every source table entry contains the following data:

- the prefix (prefix, plen), where plen is the prefix length in bits, that this entry applies to;
- the router-id of a router originating this prefix;
- a pair (seqno, metric), this source's feasibility distance.

There is one timer associated with each entry in the source table -- the source garbage-collection timer. It is initialised to a time on the order of minutes and reset as specified in Section 3.7.3.

3.2.6. The Route Table

The route table contains the routes known to this node. It is indexed by triples of the form (prefix, plen, neighbour), and every route table entry contains the following data:

- the source (prefix, plen, router-id) for which this route is advertised;
- the neighbour (an entry in the neighbour table) that advertised this route;
- the metric with which this route was advertised by the neighbour, or FFFF hexadecimal (infinity) for a recently retracted route;
- the sequence number with which this route was advertised;
- the next-hop address of this route;

- a boolean flag indicating whether this route is selected, i.e., whether it is currently being used for forwarding and is being advertised.

There is one timer associated with each route table entry -- the route expiry timer. It is initialised and reset as specified in Section 3.5.3.

Note that there are two distinct (seqno, metric) pairs associated to each route: the route's distance, which is stored in the route table, and the feasibility distance, stored in the source table and shared between all routes with the same source.

3.2.7. The Table of Pending Seqno Requests

The table of pending seqno requests contains a list of seqno requests that the local node has sent (either because they have been originated locally, or because they were forwarded) and to which no reply has been received yet. This table is indexed by triples of the form (prefix, plen, router-id), and every entry in this table contains the following data:

- the prefix, plen, router-id, and seqno being requested;
- the neighbour, if any, on behalf of which we are forwarding this request;
- a small integer indicating the number of times that this request will be resent if it remains unsatisfied.

There is one timer associated with each pending seqno request; it governs both the resending of requests and their expiry.

3.3. Acknowledgments and acknowledgment requests

A Babel speaker may request that a neighbour receiving a given packet reply with an explicit acknowledgment within a given time. While the use of acknowledgment requests is optional, every Babel speaker MUST be able to reply to such a request.

An acknowledgment MUST be sent to a unicast destination. On the other hand, acknowledgment requests may be sent to either unicast or multicast destinations, in which case they request an acknowledgment from all of the receiving nodes.

When to request acknowledgments is a matter of local policy; the simplest strategy is to never request acknowledgments and to rely on periodic updates to ensure that any reachable routes are eventually

propagated throughout the routing domain. In order to improve convergence speed and reduce the amount of control traffic, acknowledgment requests MAY be used in order to reliably send urgent updates (Section 3.7.2) and retractions (Section 3.5.4), especially when the number of neighbours on a given interface is small. Since Babel is designed to deal gracefully with packet loss on unreliable media, sending all packets with acknowledgment requests is not necessary, and NOT RECOMMENDED, as the acknowledgments cause additional traffic and may force additional Address Resolution Protocol (ARP) or Neighbour Discovery (ND) exchanges.

3.4. Neighbour Acquisition

Neighbour acquisition is the process by which a Babel node discovers the set of neighbours heard over each of its interfaces and ascertains bidirectional reachability. On unreliable media, neighbour acquisition additionally provides some statistics that may be useful for link quality computation.

Before it can exchange routing information with a neighbour, a Babel node MUST create an entry for that neighbour in the neighbour table. When to do that is implementation-specific; suitable strategies include creating an entry when any Babel packet is received, or creating an entry when a Hello TLV is parsed. Similarly, in order to conserve system resources, an implementation SHOULD discard an entry when it has been unused for long enough; suitable strategies include dropping the neighbour after a timeout, and dropping a neighbour when the associated Hello histories become empty (see Appendix A.2).

3.4.1. Reverse Reachability Detection

Every Babel node sends Hello TLVs to its neighbours to indicate that it is alive, at regular or irregular intervals. Each Hello TLV carries an increasing (modulo 2^16) sequence number and an upper bound on the time interval until the next Hello of the same type (see below). If the time interval is set to 0, then the Hello TLV does not establish a new promise: the deadline carried by the previous Hello of the same type still applies to the next Hello (if the most recent scheduled Hello of the right kind was received at time t0 and carried interval i, then the previous promise of sending another Hello before time t0 + i still holds). We say that a Hello is "scheduled" if it carries a non-zero interval, and "unscheduled" otherwise.

There are two kinds of Hellos: Multicast Hellos, which use a per-interface Hello counter (the Multicast Hello seqno), and Unicast Hellos, which use a per-neighbour counter (the Unicast Hello seqno). A Multicast Hello with a given seqno MUST be sent to all neighbours

on a given interface, either by sending it to a multicast address or by sending it to one unicast address per neighbour (hence, the term "Multicast Hello" is a slight misnomer). A Unicast Hello carrying a given seqno should normally be sent to just one neighbour (over unicast), since the sequence numbers of different neighbours are not in general synchronised.

Multicast Hellos sent over multicast can be used for neighbour discovery; hence, a node SHOULD send periodic (scheduled) Multicast Hellos unless neighbour discovery is performed by means outside of the Babel protocol. A node MAY send Unicast Hellos or unscheduled Hellos of either kind for any reason, such as reducing the amount of multicast traffic or improving reliability on link technologies with poor support for link-layer multicast.

A node MAY send a scheduled Hello ahead of time. A node MAY change its scheduled Hello interval. The Hello interval MAY be decreased at any time; it MAY be increased immediately before sending a Hello TLV, but SHOULD NOT be increased at other times. (Equivalently, a node SHOULD send a scheduled Hello immediately after increasing its Hello interval.)

How to deal with received Hello TLVs and what statistics to maintain are considered local implementation matters; typically, a node will maintain some sort of history of recently received Hellos. An example of a suitable algorithm is described in Appendix A.1.

After receiving a Hello, or determining that it has missed one, the node recomputes the association's cost (Section 3.4.3) and runs the route selection procedure (Section 3.6).

3.4.2. Bidirectional Reachability Detection

In order to establish bidirectional reachability, every node sends periodic IHU ("I Heard You") TLVs to each of its neighbours. Since IHUs carry an explicit interval value, they MAY be sent less often than Hellos in order to reduce the amount of routing traffic in dense networks; in particular, they SHOULD be sent less often than Hellos over links with little packet loss. While IHUs are conceptually unicast, they MAY be sent to a multicast address in order to avoid an ARP or Neighbour Discovery exchange and to aggregate multiple IHUs into a single packet.

In addition to the periodic IHUs, a node MAY, at any time, send an unscheduled IHU packet. It MAY also, at any time, decrease its IHU interval, and it MAY increase its IHU interval immediately before sending an IHU, but SHOULD NOT increase it at any other time.

(Equivalently, a node SHOULD send an extra IHU immediately after increasing its Hello interval.)

Every IHU TLV contains two pieces of data: the link's rxcost (reception cost) from the sender's perspective, used by the neighbour for computing link costs (Section 3.4.3), and the interval between periodic IHU packets. A node receiving an IHU sets the value of the txcost (transmission cost) maintained in the neighbour table to the value contained in the IHU, and resets the IHU timer associated to this neighbour to a small multiple of the interval value received in the IHU (see "IHU Hold time" in Appendix B for suggested values). When a neighbour's IHU timer expires, the neighbour's txcost is set to infinity.

After updating a neighbour's txcost, the receiving node recomputes the neighbour's cost (Section 3.4.3) and runs the route selection procedure (Section 3.6).

3.4.3. Cost Computation

A neighbourship association's link cost is computed from the values maintained in the neighbour table: the statistics kept in the neighbour table about the reception of Hellos, and the txcost computed from received IHU packets.

For every neighbour, a Babel node computes a value known as this neighbour's rxcost. This value is usually derived from the Hello history, which may be combined with other data, such as statistics maintained by the link layer. The rxcost is sent to a neighbour in each IHU.

Since nodes do not necessarily send periodic Unicast Hellos but do usually send periodic Multicast Hellos (Section 3.4.1), a node SHOULD use an algorithm that yields a finite rxcost when only Multicast Hellos are received, unless interoperability with nodes that only send Multicast Hellos is not required.

How the txcost and rxcost are combined in order to compute a link's cost is a matter of local policy; as far as Babel's correctness is concerned, only the following conditions MUST be satisfied:

- the cost is strictly positive;
- if no Hello TLVs of either kind were received recently, then the cost is infinite;
- if the txcost is infinite, then the cost is infinite.

See Appendix A.2 for RECOMMENDED strategies for computing a link's cost.

3.5. Routing Table Maintenance

Conceptually, a Babel update is a quintuple (prefix, plen, router-id, seqno, metric), where (prefix, plen) is the prefix for which a route is being advertised, router-id is the router-id of the router originating this update, seqno is a nondecreasing (modulo 2^16) integer that carries the originating router seqno, and metric is the announced metric.

Before being accepted, an update is checked against the feasibility condition (Section 3.5.1), which ensures that the route does not create a routing loop. If the feasibility condition is not satisfied, the update is either ignored or prevents the route from being selected, as described in Section 3.5.3. If the feasibility condition is satisfied, then the update cannot possibly cause a routing loop.

3.5.1. The Feasibility Condition

The feasibility condition is applied to all received updates. The feasibility condition compares the metric in the received update with the metrics of the updates previously sent by the receiving node; updates that fail the feasibility condition, and therefore have metrics large enough to cause a routing loop, are either ignored or prevent the resulting route from being selected.

A feasibility distance is a pair (seqno, metric), where seqno is an integer modulo 2^16 and metric is a positive integer. Feasibility distances are compared lexicographically, with the first component inverted: we say that a distance (seqno, metric) is strictly better than a distance (seqno', metric'), written

(seqno, metric) < (seqno', metric')

when

seqno > seqno' or (seqno = seqno' and metric < metric')

where sequence numbers are compared modulo 2^16.

Given a source (prefix, plen, router-id), a node's feasibility distance for this source is the minimum, according to the ordering defined above, of the distances of all the finite updates ever sent by this particular node for the prefix (prefix, plen) and the given

router-id. Feasibility distances are maintained in the source table, the exact procedure is given in Section 3.7.3.

A received update is feasible when either it is a retraction (its metric is FFFF hexadecimal), or the advertised distance is strictly better, in the sense defined above, than the feasibility distance for the corresponding source. More precisely, a route advertisement carrying the quintuple (prefix, plen, router-id, seqno, metric) is feasible if one of the following conditions holds:

- metric is infinite; or
- no entry exists in the source table indexed by (prefix, plen, router-id); or
- an entry (prefix, plen, router-id, seqno', metric') exists in the source table, and either
  - seqno' < seqno or
  - seqno = seqno' and metric < metric'.

Note that the feasibility condition considers the metric advertised by the neighbour, not the route's metric; hence, a fluctuation in a neighbour's cost cannot render a selected route unfeasible. Note further that retractions (updates with infinite metric) are always feasible, since they cannot possibly cause a routing loop.

3.5.2. Metric Computation

A route's metric is computed from the metric advertised by the neighbour and the neighbour's link cost. Just like cost computation, metric computation is considered a local policy matter; as far as Babel is concerned, the function M(c, m) used for computing a metric from a locally computed link cost c and the metric m advertised by a neighbour MUST only satisfy the following conditions:

- if c is infinite, then M(c, m) is infinite;
- M is strictly monotonic: M(c, m) > m.

Additionally, the metric SHOULD satisfy the following condition:

- M is left-distributive: if m <= m', then M(c, m) <= M(c, m').

While strict monotonicity is essential to the integrity of the network (persistent routing loops may arise if it is not satisfied), left distributivity is not: if it is not satisfied, Babel will still

converge to a loop-free configuration, but might not reach a global optimum (in fact, a global optimum may not even exist).

The conditions above are easily satisfied by using the additive metric, i.e., by defining M(c, m) = c + m. Since the additive metric is useful with a large range of cost computation strategies, it is the RECOMMENDED default. See also Appendix C, which describes a technique that makes it possible to tweak the values of individual metrics without running the risk of creating routing loops.

3.5.3. Route Acquisition

When a Babel node receives an update (prefix, plen, router-id, seqno, metric) from a neighbour neigh, it checks whether it already has a route table entry indexed by (prefix, plen, neigh).

If no such entry exists:

- if the update is unfeasible, it MAY be ignored;
- if the metric is infinite (the update is a retraction of a route we do not know about), the update is ignored;
- otherwise, a new entry is created in the route table, indexed by (prefix, plen, neigh), with source equal to (prefix, plen, router-id), seqno equal to seqno and an advertised metric equal to the metric carried by the update.

If such an entry exists:

- if the entry is currently selected, the update is unfeasible, and the router-id of the update is equal to the router-id of the entry, then the update MAY be ignored;
- otherwise, the entry's sequence number, advertised metric, metric, and router-id are updated and, if the advertised metric is not infinite, the route's expiry timer is reset to a small multiple of the Interval value included in the update (see "Route Hold time" in Appendix B for suggested values). If the update is unfeasible, then the (now unfeasible) entry MUST be immediately unselected. If the update caused the router-id of the entry to change, an update (possibly a retraction) MUST be sent in a timely manner as described in Section 3.7.2.

Note that the route table may contain unfeasible routes, either because they were created by an unfeasible update or due to a metric fluctuation. Such routes are never selected, since they are not known to be loop-free; should all the feasible routes become

unusable, however, the unfeasible routes can be made feasible and therefore possible to select by sending requests along them (see Section 3.8.2).

When a route's expiry timer triggers, the behaviour depends on whether the route's metric is finite. If the metric is finite, it is set to infinity and the expiry timer is reset. If the metric is already infinite, the route is flushed from the route table.

After the route table is updated, the route selection procedure (Section 3.6) is run.

3.5.4. Hold Time

When a prefix P is retracted, because all routes are unfeasible or have an infinite metric (whether due to the expiry timer or to other reasons), and a shorter prefix P' that covers P is reachable, P' cannot in general be used for routing packets destined to P without running the risk of creating a routing loop (Section 2.8).

To avoid this issue, whenever a prefix P is retracted, a route table entry with infinite metric is maintained as described in Section 3.5.3 above. As long as this entry is maintained, packets destined to an address within P MUST NOT be forwarded by following a route for a shorter prefix. This entry is removed as soon as a finite-metric update for prefix P is received and the resulting route selected. If no such update is forthcoming, the infinite metric entry SHOULD be maintained at least until it is guaranteed that no neighbour has selected the current node as next-hop for prefix P. This can be achieved by either:

- waiting until the route's expiry timer has expired (Section 3.5.3);
- sending a retraction with an acknowledgment request (Section 3.3) to every reachable neighbour that has not explicitly retracted prefix P, and waiting for all acknowledgments.

The former option is simpler and ensures that at that point, any routes for prefix P pointing at the current node have expired. However, since the expiry time can be as high as a few minutes, doing that prevents automatic aggregation by creating spurious black-holes for aggregated routes. The latter option is RECOMMENDED as it dramatically reduces the time for which a prefix is unreachable in the presence of aggregated routes.

3.6. Route Selection

Route selection is the process by which a single route for a given prefix is selected to be used for forwarding packets and to be re-advertised to a node's neighbours.

Babel is designed to allow flexible route selection policies. As far as the algorithm's correctness is concerned, the route selection policy MUST only satisfy the following properties:

- a route with infinite metric (a retracted route) is never selected;
- an unfeasible route is never selected.

Babel nodes using different route selection strategies will interoperate and not create routing loops as long as these two properties hold.

Route selection MUST NOT take seqnos into account: a route MUST NOT be preferred just because it carries a higher (more recent) seqno. Doing otherwise would cause route oscillation while a new seqno propagates across the network, and might create persistent blackholes if the metric being used is not left-distributive (Section 3.5.2).

The obvious route selection strategy is to pick, for every destination, the feasible route with minimal metric. When all metrics are stable, this approach ensures convergence to a tree of shortest paths (assuming that the metric is left-distributive, see Section 3.5.2). There are two reasons, however, why this strategy may lead to instability in the presence of continuously varying metrics. First, if two parallel routes oscillate around a common value, then the smallest metric strategy will keep switching between the two. Second, when a route is selected, congestion along it increases, which might increase packet loss, which in turn could cause its metric to increase; this is a feedback loop, of the kind that is prone to causing persistent oscillations.

In order to limit these kinds of instabilities, a route selection strategy SHOULD include some form of hysteresis, i.e., be sensitive to a route's history: if a route is currently selected, then the strategy should only switch to a different route if the latter has been consistently good for some period of time. A RECOMMENDED hysteresis algorithm is given in Appendix A.3.

After the route selection procedure is run, triggered updates (Section 3.7.2) and requests (Section 3.8.2) are sent.

3.7. Sending Updates

A Babel speaker advertises to its neighbours its set of selected routes. Normally, this is done by sending one or more multicast packets containing Update TLVs on all of its connected interfaces; however, on link technologies where multicast is significantly more expensive than unicast, a node MAY choose to send multiple copies of updates in unicast packets, especially when the number of neighbours is small.

Additionally, in order to ensure that any black-holes are reliably cleared in a timely manner, a Babel node may send retractions (updates with an infinite metric) for any recently retracted prefixes.

If an update is for a route injected into the Babel domain by the local node (e.g., it carries the address of a local interface, the prefix of a directly attached network, or a prefix redistributed from a different routing protocol), the router-id is set to the local node's router-id, the metric is set to some arbitrary finite value (typically 0), and the seqno is set to the local router's sequence number.

If an update is for a route learned from another Babel speaker, the router-id and sequence number are copied from the route table entry, and the metric is computed as specified in Section 3.5.2.

3.7.1. Periodic Updates

Every Babel speaker MUST advertise each of its selected routes on every interface, at least once every Update interval. Since Babel doesn't suffer from routing loops (there is no "counting to infinity") and relies heavily on triggered updates (Section 3.7.2), this full dump only needs to happen infrequently (see Appendix B for suggested intervals).

3.7.2. Triggered Updates

In addition to periodic routing updates, a Babel speaker sends unscheduled, or triggered, updates in order to inform its neighbours of a significant change in the network topology.

A change of router-id for the selected route to a given prefix may be indicative of a routing loop in formation; hence, whenever it changes the selected router-id for a given destination, a node MUST send an update as an urgent TLV (as defined in Section 3.1). Additionally, it SHOULD make a reasonable attempt at ensuring that all reachable neighbours receive this update.

There are two strategies for ensuring that. If the number of neighbours is small, then it is reasonable to send the update together with an acknowledgment request; the update is resent until all neighbours have acknowledged the packet, up to some number of times. If the number of neighbours is large, however, requesting acknowledgments from all of them might cause a non-negligible amount of network traffic; in that case, it may be preferable to simply repeat the update some reasonable number of times (say, 3 for wireless and 2 for wired links). The number of copies MUST NOT exceed 5, and the copies SHOULD be separated by a small delay, as described in Section 3.1.

A route retraction is somewhat less worrying: if the route retraction doesn't reach all neighbours, a black-hole might be created, which, unlike a routing loop, does not endanger the integrity of the network. When a route is retracted, a node SHOULD send a triggered update and SHOULD make a reasonable attempt at ensuring that all neighbours receive this retraction.

Finally, a node MAY send a triggered update when the metric for a given prefix changes in a significant manner, due to a received update, because a link's cost has changed, or because a different next hop has been selected. A node SHOULD NOT send triggered updates for other reasons, such as when there is a minor fluctuation in a route's metric, when the selected next hop changes without inducing a significant change to the route's metric, or to propagate a new sequence number (except to satisfy a request, as specified in Section 3.8).

3.7.3. Maintaining Feasibility Distances

Before sending an update (prefix, plen, router-id, seqno, metric) with finite metric (i.e., not a route retraction), a Babel node updates the feasibility distance maintained in the source table. This is done as follows.

If no entry indexed by (prefix, plen, router-id) exists in the source table, then one is created with value (prefix, plen, router-id, seqno, metric).

If an entry (prefix, plen, router-id, seqno', metric') exists, then it is updated as follows:

- if seqno > seqno', then seqno' := seqno, metric' := metric;
- if seqno = seqno' and metric' > metric, then metric' := metric;
- otherwise, nothing needs to be done.

The garbage-collection timer for the entry is then reset. Note that the feasibility distance is not updated and the garbage-collection timer is not reset when a retraction (an update with infinite metric) is sent.

When the garbage-collection timer expires, the entry is removed from the source table.

3.7.4. Split Horizon

When running over a transitive, symmetric link technology, e.g., a point-to-point link or a wired LAN technology such as Ethernet, a Babel node SHOULD use an optimisation known as split horizon. When split horizon is used on a given interface, a routing update for prefix P is not sent on the particular interface over which the selected route towards prefix P was learnt.

Split horizon SHOULD NOT be applied to an interface unless the interface is known to be symmetric and transitive; in particular, split horizon is not applicable to decentralised wireless link technologies (e.g., IEEE 802.11 in ad hoc mode) when routing updates are sent over multicast.

3.8. Explicit Requests

In normal operation, a node's route table is populated by the regular and triggered updates sent by its neighbours. Under some circumstances, however, a node sends explicit requests in order to cause a resynchronisation with the source after a mobility event or to prevent a route from spuriously expiring.

The Babel protocol provides two kinds of explicit requests: route requests, which simply request an update for a given prefix, and seqno requests, which request an update for a given prefix with a specific sequence number. The former are never forwarded; the latter are forwarded if they cannot be satisfied by the receiver.

3.8.1. Handling Requests

Upon receiving a request, a node either forwards the request or sends an update in reply to the request, as described in the following sections. If this causes an update to be sent, the update is either sent to a multicast address on the interface on which the request was received, or to the unicast address of the neighbour that sent the request.

The exact behaviour is different for route requests and seqno requests.

3.8.1.1. Route Requests

When a node receives a route request for a given prefix, it checks its route table for a selected route to this exact prefix. If such a route exists, it MUST send an update (over unicast or over multicast); if such a route does not exist, it MUST send a retraction for that prefix.

When a node receives a wildcard route request, it SHOULD send a full route table dump. Full route dumps SHOULD be rate-limited, especially if they are sent over multicast.

3.8.1.2. Seqno Requests

When a node receives a seqno request for a given router-id and sequence number, it checks whether its route table contains a selected entry for that prefix. If a selected route for the given prefix exists, it has finite metric, and either the router-ids are different or the router-ids are equal and the entry's sequence number is no smaller (modulo 2^16) than the requested sequence number, the node MUST send an update for the given prefix. If the router-ids match but the requested seqno is larger (modulo 2^16) than the route entry's, the node compares the router-id against its own router-id. If the router-id is its own, then it increases its sequence number by 1 (modulo 2^16) and sends an update. A node MUST NOT increase its sequence number by more than 1 in reaction to a single seqno request.

Otherwise, if the requested router-id is not its own, the received node consults the hop count field of the request. If the hop count is 2 or more, and the node is advertising the prefix to its neighbours, the node selects a neighbour to forward the request to as follows:

- if the node has one or more feasible routes toward the requested prefix with a next hop that is not the requesting node, then the node MUST forward the request to the next hop of one such route;

- otherwise, if the node has one or more (not feasible) routes to the requested prefix with a next hop that is not the requesting node, then the node SHOULD forward the request to the next hop of one such route.

In order to actually forward the request, the node decrements the hop count and sends the request in a unicast packet destined to the selected neighbour. The forwarded request SHOULD be sent as an urgent TLV (as defined in Section 3.1).

A node SHOULD maintain a list of recently forwarded seqno requests and forward the reply (an update with a seqno sufficiently large to satisfy the request) as an urgent TLV (as defined in Section 3.1). A node SHOULD compare every incoming seqno request against its list of recently forwarded seqno requests and avoid forwarding it if it is redundant (i.e., if it has recently sent a request with the same prefix, router-id and a seqno that is not smaller modulo 2^16).

Since the request-forwarding mechanism does not necessarily obey the feasibility condition, it may get caught in routing loops; hence, requests carry a hop count to limit the time during which they remain in the network. However, since requests are only ever forwarded as unicast packets, the initial hop count need not be kept particularly low, and performing an expanding horizon search is not necessary. A single request MUST NOT be duplicated: it MUST NOT be forwarded to a multicast address, and it MUST NOT be forwarded to multiple neighbours. However, if a seqno request is resent by its originator, the subsequent copies may be forwarded to a different neighbour than the initial one.

3.8.2. Sending Requests

A Babel node MAY send a route or seqno request at any time, to a multicast or a unicast address; there is only one case when originating requests is required (Section 3.8.2.1).

3.8.2.1. Avoiding Starvation

When a route is retracted or expires, a Babel node usually switches to another feasible route for the same prefix. It may be the case, however, that no such routes are available.

A node that has lost all feasible routes to a given destination but still has unexpired unfeasible routes to that destination MUST send a seqno request; if it doesn't have any such routes, it MAY still send a seqno request. The router-id of the request is set to the router-id of the route that it has just lost, and the requested seqno is the value contained in the source table plus 1. The request carries a hop count, which is used as a last-resort mechanism to ensure that it eventually vanishes from the network; it MAY be set to any value that is larger than the diameter of the network (64 is a suitable default value).

If the node has any (unfeasible) routes to the requested destination, then it MUST send the request to at least one of the next-hop neighbours that advertised these routes, and SHOULD send it to all of them; in any case, it MAY send the request to any other neighbours, whether they advertise a route to the requested destination or not.

A simple implementation strategy is therefore to unconditionally multicast the request over all interfaces.

Similar requests will be sent by other nodes that are affected by the route's loss. If the network is still connected, and assuming no packet loss, then at least one of these requests will be forwarded to the source, resulting in a route being advertised with a new sequence number. (Due to duplicate suppression, only a small number of such requests are expected to actually reach the source.)

In order to compensate for packet loss, a node SHOULD repeat such a request a small number of times if no route becomes feasible within a short time (see "Request Timeout" in Appendix B for suggested values). In the presence of heavy packet loss, however, all such requests might be lost; in that case, the mechanism in the next section will eventually ensure that a new seqno is received.

3.8.2.2. Dealing with Unfeasible Updates

When a route's metric increases, a node might receive an unfeasible update for a route that it has currently selected. As specified in Section 3.5.1, the receiving node will either ignore the update or unselect the route.

In order to keep routes from spuriously expiring because they have become unfeasible, a node SHOULD send a unicast seqno request when it receives an unfeasible update for a route that is currently selected. The requested sequence number is computed from the source table as in Section 3.8.2.1 above.

Additionally, since metric computation does not necessarily coincide with the delay in propagating updates, a node might receive an unfeasible update from a currently unselected neighbour that is preferable to the currently selected route (e.g., because it has a much smaller metric); in that case, the node SHOULD send a unicast seqno request to the neighbour that advertised the preferable update.

3.8.2.3. Preventing Routes from Expiring

In normal operation, a route's expiry timer never triggers: since a route's hold time is computed from an explicit interval included in Update TLVs, a new update (possibly a retraction) should arrive in time to prevent a route from expiring.

In the presence of packet loss, however, it may be the case that no update is successfully received for an extended period of time, causing a route to expire. In order to avoid such spurious expiry, shortly before a selected route expires, a Babel node SHOULD send a

unicast route request to the neighbour that advertised this route; since nodes always send either updates or retractions in response to non-wildcard route requests (Section 3.8.1.1), this will usually result in the route being either refreshed or retracted.

4. Protocol Encoding

A Babel packet MUST be sent as the body of a UDP datagram, with network-layer hop count set to 1, destined to a well-known multicast address or to a unicast address, over IPv4 or IPv6; in the case of IPv6, these addresses are link-local. Both the source and destination UDP port are set to a well-known port number. A Babel packet MUST be silently ignored unless its source address is either a link-local IPv6 address or an IPv4 address belonging to the local network, and its source port is the well-known Babel port. It MAY be silently ignored if its destination address is a global IPv6 address.

In order to minimise the number of packets being sent while avoiding lower-layer fragmentation, a Babel node SHOULD maximise the size of the packets it sends, up to the outgoing interface's MTU adjusted for lower-layer headers (28 octets for UDP over IPv4, 48 octets for UDP over IPv6). It MUST NOT send packets larger than the attached interface's MTU adjusted for lower-layer headers or 512 octets, whichever is larger, but not exceeding 2^16 - 1 adjusted for lower-layer headers. Every Babel speaker MUST be able to receive packets that are as large as any attached interface's MTU adjusted for lower-layer headers or 512 octets, whichever is larger. Babel packets MUST NOT be sent in IPv6 Jumbograms [RFC2675].

4.1. Data Types

4.1.1. Representation of integers

All multi-octet fields that represent integers are encoded with the most significant octet first (in Big-Endian format [IEN137], also called Network Order). The base protocol only carries unsigned values; if an extension needs to carry signed values, it will need to specify their encoding (e.g., two's complement).

4.1.2. Interval

Relative times are carried as 16-bit values specifying a number of centiseconds (hundredths of a second). This allows times up to roughly 11 minutes with a granularity of 10ms, which should cover all reasonable applications of Babel (see also Appendix B).

4.1.3. Router-Id

A router-id is an arbitrary 8-octet value. A router-id MUST NOT consist of either all binary zeroes (0000000000000000 hexadecimal) or all binary ones (FFFFFFFFFFFFFFFF hexadecimal).

4.1.4. Address

Since the bulk of the protocol is taken by addresses, multiple ways of encoding addresses are defined. Additionally, within Update TLVs a common subnet prefix may be omitted when multiple addresses are sent in a single packet -- this is known as address compression (Section 4.6.9).

Address encodings:

- AE 0: wildcard address. The value is 0 octets long.
- AE 1: IPv4 address. Compression is allowed. 4 octets or less.
- AE 2: IPv6 address. Compression is allowed. 16 octets or less.
- AE 3: link-local IPv6 address. Compression is not allowed. The value is 8 octets long, a prefix of fe80::/64 is implied.

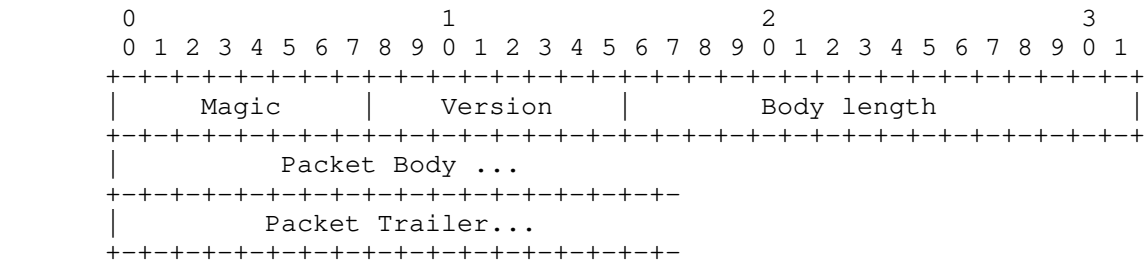
The address family associated to an address encoding is either IPv4 or IPv6; it is undefined for AE 0, IPv4 for AE 1, and IPv6 for AEs 2 and 3.

4.1.5. Prefixes

A network prefix is encoded just like a network address, but it is stored in the smallest number of octets that are enough to hold the significant bits (up to the prefix length).

4.2. Packet Format

A Babel packet consists of a 4-octet header, followed by a sequence of TLVs (the packet body), optionally followed by a second sequence of TLVs (the packet trailer). The format is designed to be extensible; see Appendix D for extensibility considerations.



Fields :

Magic The arbitrary but carefully chosen value 42 (decimal); packets with a first octet different from 42 MUST be silently ignored.

Version This document specifies version 2 of the Babel protocol. Packets with a second octet different from 2 MUST be silently ignored.

Body length The length in octets of the body following the packet header (excluding the Magic, Version and Body length fields, and excluding the packet trailer).

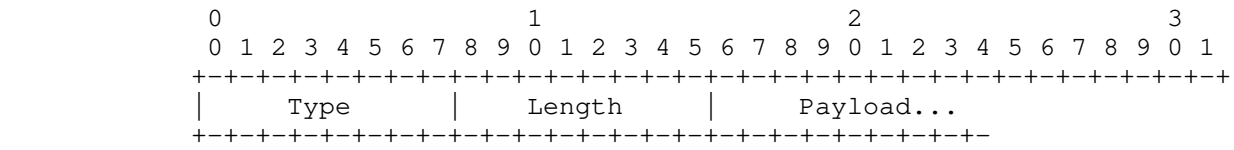
Packet Body The packet body; a sequence of TLVs.

Packet Trailer The packet trailer; another sequence of TLVs.

The packet body and trailer are both sequences of TLVs. The packet body is the normal place to store TLVs; the packet trailer only contains specialised TLVs that do not need to be protected by cryptographic security mechanisms. When parsing the trailer, the receiver MUST ignore any TLV unless its definition explicitly states that it is allowed to appear there. Among the TLVs defined in this document, only Pad1 and PadN are allowed in the trailer; since these TLVs are ignored in any case, an implementation MAY silently ignore the packet trailer without even parsing it, unless it implements at least one protocol extension that defines TLVs that are allowed to appear in the trailer.

4.3. TLV Format

With the exception of Pad1, all TLVs have the following structure:



Fields :

Type The type of the TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

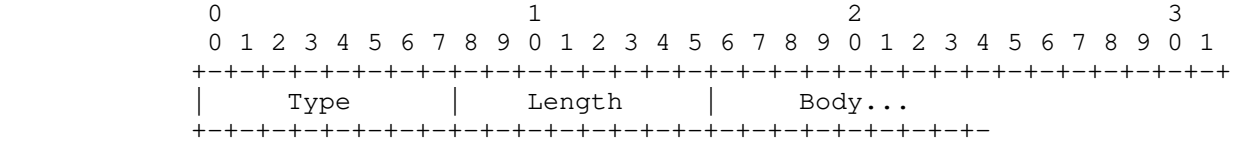
Payload The TLV payload, which consists of a body and, for selected TLV types, an optional list of sub-TLVs.

TLVs with an unknown type value MUST be silently ignored.

4.4. Sub-TLV Format

Every TLV carries an explicit length in its header; however, most TLVs are self-terminating, in the sense that it is possible to determine the length of the body without reference to the explicit Length field. If a TLV has a self-terminating format, then the space between the natural size of the TLV and the size announced in the Length field may be used to store a sequence of sub-TLVs.

Sub-TLVs have the same structure as TLVs. With the exception of Pad1, all TLVs have the following structure:



Fields :

Type The type of the sub-TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

Body The sub-TLV body, the interpretation of which depends on both the type of the sub-TLV and the type of the TLV within which it is embedded.

The most-significant bit of the sub-TLV type (the bit with value 80 hexadecimal), is called the mandatory bit; in other words, sub-TLV types 128 through 255 have the mandatory bit set. This bit indicates how to handle unknown sub-TLVs. If the mandatory bit is not set, then an unknown sub-TLV MUST be silently ignored, and the rest of the TLV is processed normally. If the mandatory bit is set, then the whole enclosing TLV MUST be silently ignored (except for updating the parser state by a Router-Id, Next-Hop or Update TLV, as described in the next section).

4.5. Parser state and encoding of updates

In a large network, the bulk of Babel traffic consists of route updates; hence, some care has been given to encoding them efficiently. The data conceptually contained in an update (Section 3.5) is split into three pieces:

- the prefix, seqno and metric are contained in the Update TLV itself (Section 4.6.9);
- the router-id is taken from Router-Id TLV that precedes the Update TLV, and may be shared among multiple Update TLVs (Section 4.6.7);
- the next hop is taken either from the source-address of the network-layer packet that contains the Babel packet, or from an explicit Next-Hop TLV (Section 4.6.8).

In addition to the above, an Update TLV can omit a prefix of the prefix being announced, which is then extracted from the preceding Update TLV in the same address family (IPv4 or IPv6). Finally, as a special optimisation for the case when a router-id coincides with the interface-id part of an IPv6 address, Router-ID TLV itself may be omitted and the router-id derived from the low-order bits of the advertised prefix (Section 4.6.9).

In order to implement these compression techniques, Babel uses a stateful parser: a TLV may refer to data from a previous TLV. The parser state consists of the following pieces of data:

- for each address encoding that allows compression, the current default prefix; this is undefined at the start of the packet, and is updated by each Update TLV with the Prefix flag set (Section 4.6.9);
- for each address family (IPv4 or IPv6), the current next-hop; this is the source address of the enclosing packet for the matching address family at the start of a packet, and is updated by each Next-Hop TLV (Section 4.6.8);

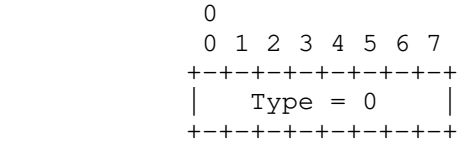
- o the current router-id; this is undefined at the start of the packet, and is updated by each Router-ID TLV (Section 4.6.7) and by each Update TLV with Router-Id flag set.

Since the parser state must be identical across implementations, it is updated before checking for mandatory sub-TLVs: parsing a TLV MUST update the parser state even if the TLV is otherwise ignored due to an unknown mandatory sub-TLV or for any other reason.

None of the TLVs that modify the parser state are allowed in the packet trailer; hence, an implementation may choose to use a dedicated stateless parser to parse the packet trailer.

4.6. Details of Specific TLVs

4.6.1. Pad1

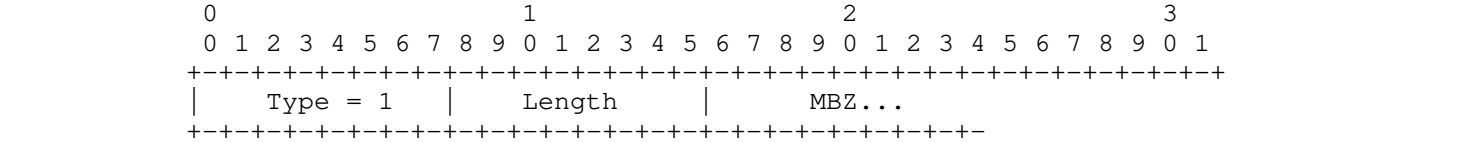


Fields :

Type Set to 0 to indicate a Pad1 TLV.

This TLV is silently ignored on reception. It is allowed in the packet trailer.

4.6.2. PadN



Fields :

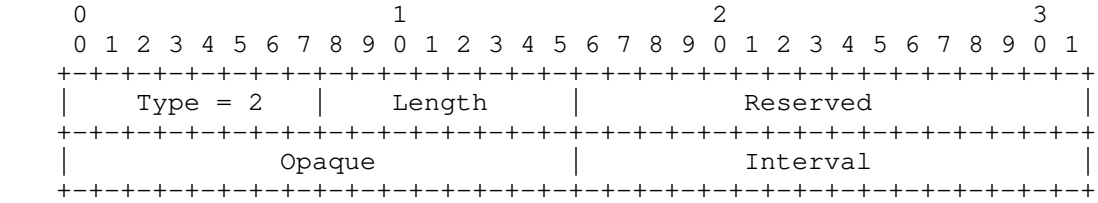
Type Set to 1 to indicate a PadN TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

MBZ Must be zero, set to 0 on transmission.

This TLV is silently ignored on reception. It is allowed in the packet trailer.

4.6.3. Acknowledgment Request



This TLV requests that the receiver send an Acknowledgment TLV within the number of centiseconds specified by the Interval field.

Fields :

Type Set to 2 to indicate an Acknowledgment Request TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

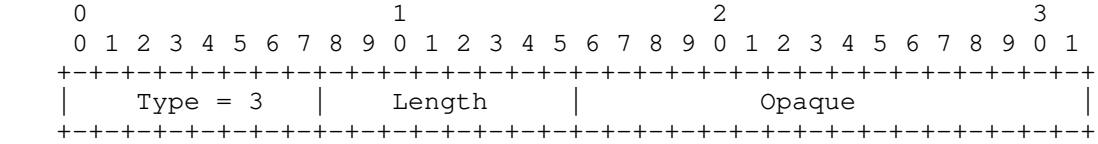
Reserved Sent as 0 and MUST be ignored on reception.

Opaque An arbitrary value that will be echoed in the receiver's Acknowledgment TLV.

Interval A time interval in centiseconds after which the sender will assume that this packet has been lost. This MUST NOT be 0. The receiver MUST send an Acknowledgment TLV before this time has elapsed (with a margin allowing for propagation time).

This TLV is self-terminating, and allows sub-TLVs.

4.6.4. Acknowledgment



This TLV is sent by a node upon receiving an Acknowledgment Request.

Fields :

Type Set to 3 to indicate an Acknowledgment TLV.

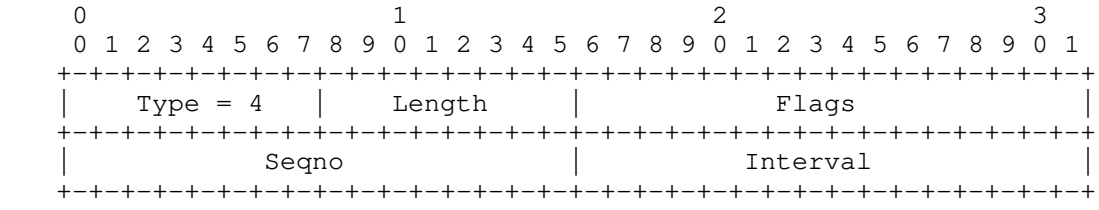
Length The length of the body in octets, exclusive of the Type and Length fields.

Opaque Set to the Opaque value of the Acknowledgment Request that prompted this Acknowledgment.

Since Opaque values are not globally unique, this TLV MUST be sent to a unicast address.

This TLV is self-terminating, and allows sub-TLVs.

4.6.5. Hello



This TLV is used for neighbour discovery and for determining a neighbour's reception cost.

Fields :

Type Set to 4 to indicate a Hello TLV.

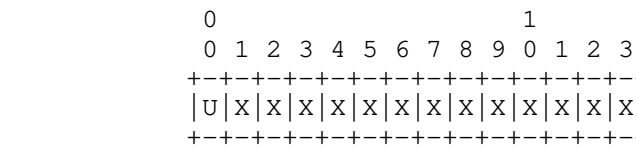
Length The length of the body in octets, exclusive of the Type and Length fields.

Flags The individual bits of this field specify special handling of this TLV (see below).

Seqno If the Unicast flag is set, this is the value of the sending node's outgoing Unicast Hello seqno for this neighbour. Otherwise, it is the sending node's outgoing Multicast Hello seqno for this interface.

Interval If non-zero, this is an upper bound, expressed in centiseconds, on the time after which the sending node will send a new scheduled Hello TLV with the same setting of the Unicast flag. If this is 0, then this Hello represents an unscheduled Hello, and doesn't carry any new information about times at which Hellos are sent.

The Flags field is interpreted as follows:



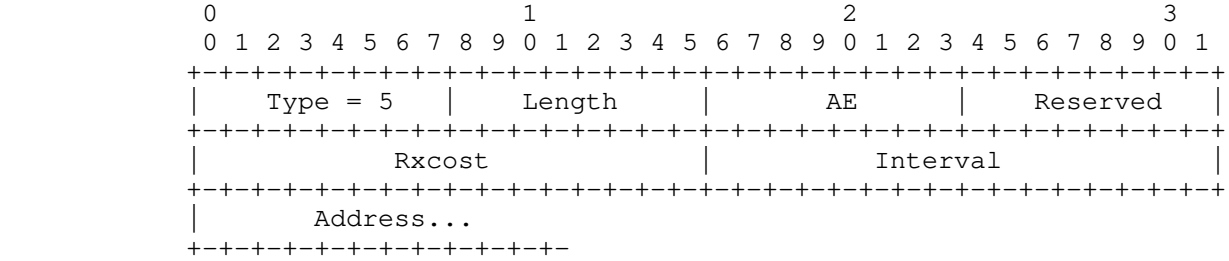
- U (Unicast) flag (8000 hexadecimal): if set, then this Hello represents a Unicast Hello, otherwise it represents a Multicast Hello;

- X: all other bits MUST be sent as 0 and silently ignored on reception.

Every time a Hello is sent, the corresponding seqno counter MUST be incremented. Since there is a single seqno counter for all the Multicast Hellos sent by a given node over a given interface, if the Unicast flag is not set, this TLV MUST be sent to all neighbors on this link, which can be achieved by sending to a multicast destination, or by sending multiple packets to the unicast addresses of all reachable neighbours. Conversely, if the Unicast flag is set, this TLV MUST be sent to a single neighbour, which can achieved by sending to a unicast destination. In order to avoid large discontinuities in link quality, multiple Hello TLVs SHOULD NOT be sent in the same packet.

This TLV is self-terminating, and allows sub-TLVs.

4.6.6. IHU



An IHU ("I Heard You") TLV is used for confirming bidirectional reachability and carrying a link's transmission cost.

Fields :

Type Set to 5 to indicate an IHU TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

AE The encoding of the Address field. This should be 1 or 3 in most cases. As an optimisation, it MAY be 0 if the TLV is sent to a unicast address, if the association is over a point-to-point link, or when bidirectional reachability is ascertained by means outside of the Babel protocol.

Reserved Sent as 0 and MUST be ignored on reception.

Rxcost The rxcost according to the sending node of the interface whose address is specified in the Address field. The value FFFF hexadecimal (infinity) indicates that this interface is unreachable.

Interval An upper bound, expressed in centiseconds, on the time after which the sending node will send a new IHU; this MUST NOT be 0. The receiving node will use this value in order to compute a hold time for this symmetric association.

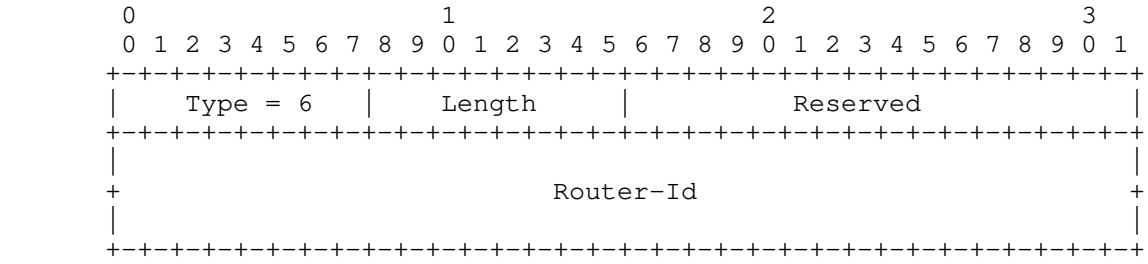
Address The address of the destination node, in the format specified by the AE field. Address compression is not allowed.

Conceptually, an IHU is destined to a single neighbour. However, IHU TLVs contain an explicit destination address, and MAY be sent to a multicast address, as this allows aggregation of IHUs destined to distinct neighbours into a single packet and avoids the need for an ARP or Neighbour Discovery exchange when a neighbour is not being used for data traffic.

IHU TLVs with an unknown value in the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.7. Router-Id



A Router-Id TLV establishes a router-id that is implied by subsequent Update TLVs, as described in Section 4.5. This TLV sets the router-id even if it is otherwise ignored due to an unknown mandatory sub-TLV.

Fields :

Type Set to 6 to indicate a Router-Id TLV.

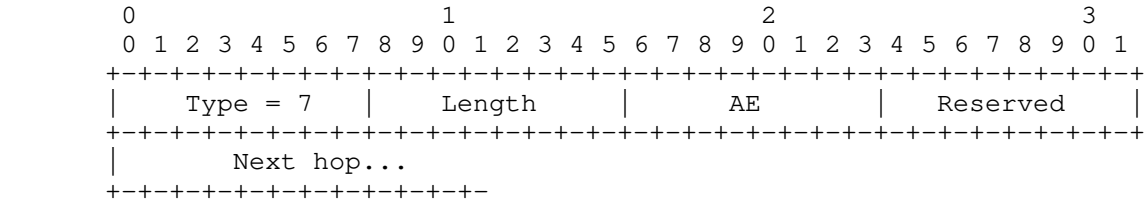
Length The length of the body in octets, exclusive of the Type and Length fields.

Reserved Sent as 0 and MUST be ignored on reception.

Router-Id The router-id for routes advertised in subsequent Update TLVs. This MUST NOT consist of all zeroes or all ones.

This TLV is self-terminating, and allows sub-TLVs.

4.6.8. Next Hop



A Next Hop TLV establishes a next-hop address for a given address family (IPv4 or IPv6) that is implied in subsequent Update TLVs, as described in Section 4.5. This TLV sets up the next-hop for subsequent Update TLVs even if it is otherwise ignored due to an unknown mandatory sub-TLV.

Fields :

Type Set to 7 to indicate a Next Hop TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

AE The encoding of the Address field. This SHOULD be 1 (IPv4) or 3 (link-local IPv6), and MUST NOT be 0.

Reserved Sent as 0 and MUST be ignored on reception.

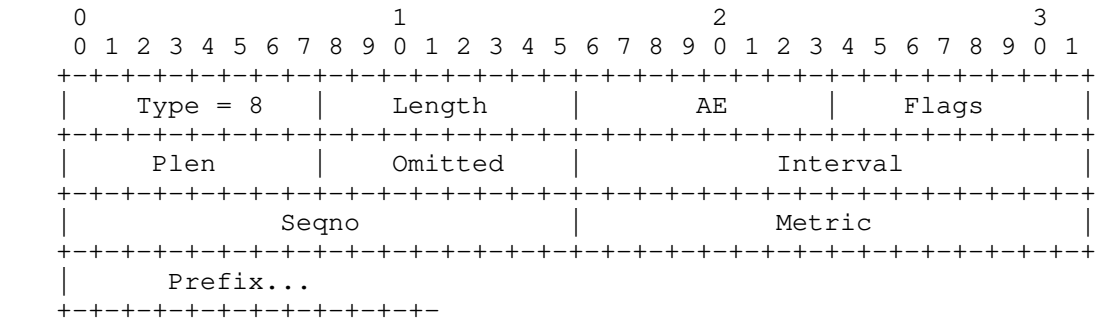
Next hop The next-hop address advertised by subsequent Update TLVs, for this address family.

When the address family matches the network-layer protocol that this packet is transported over, a Next Hop TLV is not needed: in the absence of a Next Hop TLV in a given address family, the next hop address is taken to be the source address of the packet.

Next Hop TLVs with an unknown value for the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.9. Update



An Update TLV advertises or retracts a route. As an optimisation, it can optionally have the side effect of establishing a new implied router-id and a new default prefix, as described in Section 4.5.

Fields :

Type Set to 8 to indicate an Update TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

AE The encoding of the Prefix field.

Flags The individual bits of this field specify special handling of this TLV (see below).

Plen The length in bits of the advertised prefix. If AE is 3 (link-local IPv6), Omitted MUST be 0.

Omitted The number of octets that have been omitted at the beginning of the advertised prefix and that should be taken

from a preceding Update TLV in the same address family with the Prefix flag set.

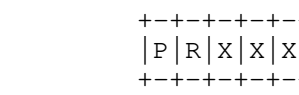
Interval An upper bound, expressed in centiseconds, on the time after which the sending node will send a new update for this prefix. This MUST NOT be 0. The receiving node will use this value to compute a hold time for the route table entry. The value FFFF hexadecimal (infinity) expresses that this announcement will not be repeated unless a request is received (Section 3.8.2.3).

Seqno The originator's sequence number for this update.

Metric The sender's metric for this route. The value FFFF hexadecimal (infinity) means that this is a route retraction.

Prefix The prefix being advertised. This field's size is (Plen/8 - Omitted) rounded upwards.

The Flags field is interpreted as follows:



- P (Prefix) flag (80 hexadecimal): if set, then this Update establishes a new default prefix for subsequent Update TLVs with a matching address encoding within the same packet, even if this TLV is otherwise ignored due to an unknown mandatory sub-TLV;

- R (Router-Id) flag (40 hexadecimal): if set, then this TLV establishes a new default router-id for this TLV and subsequent Update TLVs in the same packet, even if this TLV is otherwise ignored due to an unknown mandatory sub-TLV. This router-id is computed from the first address of the advertised prefix as follows:

  * if the length of the address is 8 octets or more, then the new router-id is taken from the 8 last octets of the address;

  * if the length of the address is smaller than 8 octets, then the new router-id consists of the required number of zero octets followed by the address, i.e., the address is stored on the right of the router-id. For example, for an IPv4 address, the router-id consists of 4 octets of zeroes followed by the IPv4 address.

- X: all other bits MUST be sent as 0 and silently ignored on reception.

The prefix being advertised by an Update TLV is computed as follows:

- the first Omitted octets of the prefix are taken from the previous Update TLV with the Prefix flag set and the same address encoding, even if it was ignored due to an unknown mandatory sub-TLV; if Omitted is not zero and there is no such TLV, then this Update MUST be ignored;

- the next (Plen/8 - Omitted) rounded upwards octets are taken from the Prefix field;

- if Plen is not a multiple of 8, then any bits beyond Plen (i.e., the low-order (8 - Plen MOD 8) bits of the last octet) are cleared;

- the remaining octets are set to 0.

If the Metric field is finite, the router-id of the originating node for this announcement is taken from the prefix advertised by this Update if the Router-Id flag is set, computed as described above. Otherwise, it is taken either from the preceding Router-Id TLV, or the preceding Update TLV with the Router-Id flag set, whichever comes last, even if that TLV is otherwise ignored due to an unknown mandatory sub-TLV; if there is no suitable TLV, then this update is ignored.

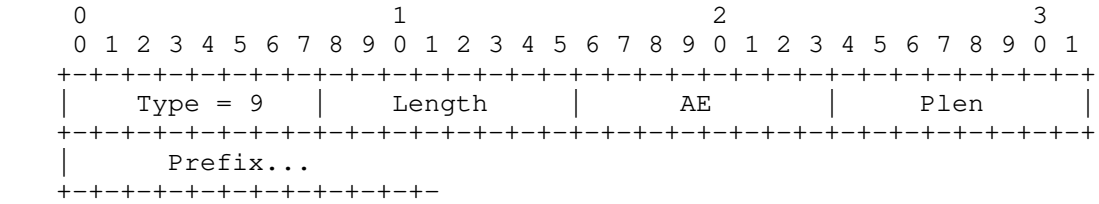
The next-hop address for this update is taken from the last preceding Next Hop TLV with a matching address family (IPv4 or IPv6) in the same packet even if it was otherwise ignored due to an unknown mandatory sub-TLV; if no such TLV exists, it is taken from the network-layer source address of this packet if it belongs to the same address family as the prefix being announced; otherwise, this Update MUST be ignored.

If the metric field is FFFF hexadecimal, this TLV specifies a retraction. In that case, the router-id, next-hop and seqno are not used. AE MAY then be 0, in which case this Update retracts all of the routes previously advertised by the sending interface. If the metric is finite, AE MUST NOT be 0; Update TLVs with finite metric and AE equal to 0 MUST be ignored. If the metric is infinite and AE is 0, Plen and Omitted MUST both be 0; Update TLVs that do not satisfy this requirement MUST be ignored.

Update TLVs with an unknown value in the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.10. Route Request



A Route Request TLV prompts the receiver to send an update for a given prefix, or a full route table dump.

Fields :

Type Set to 9 to indicate a Route Request TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

AE The encoding of the Prefix field. The value 0 specifies that this is a request for a full route table dump (a wildcard request).

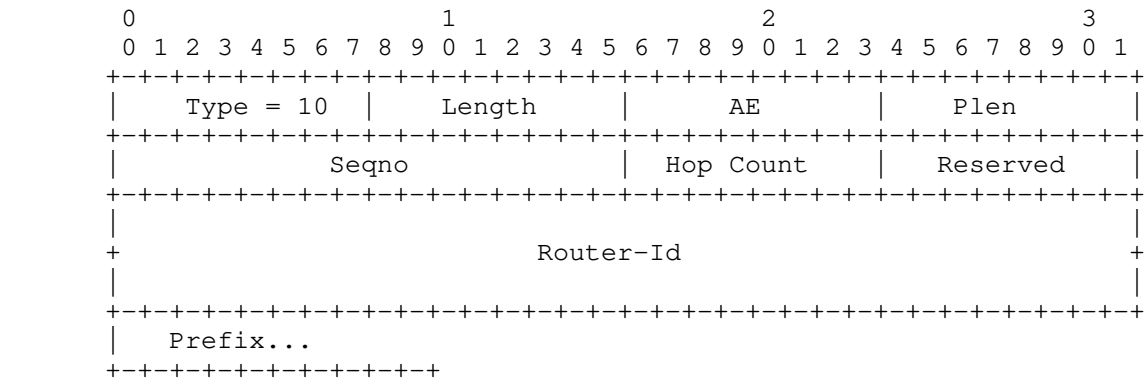
Plen The length in bits of the requested prefix.

Prefix The prefix being requested. This field's size is Plen/8 rounded upwards.

A Request TLV prompts the receiver to send an update message (possibly a retraction) for the prefix specified by the AE, Plen, and Prefix fields, or a full dump of its route table if AE is 0 (in which case Plen must be 0 and Prefix is of length 0). A Request TLV with AE set to 0 and Plen not set to 0 MUST be ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.11. Seqno Request



A Seqno Request TLV prompts the receiver to send an Update for a given prefix with a given sequence number, or to forward the request further if it cannot be satisfied locally.

Fields :

Type Set to 10 to indicate a Seqno Request TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

AE The encoding of the Prefix field. This MUST NOT be 0.

Plen The length in bits of the requested prefix.

Seqno The sequence number that is being requested.

Hop Count The maximum number of times that this TLV may be forwarded, plus 1. This MUST NOT be 0.

Reserved Sent as 0 and MUST be ignored on reception.

Router-Id The Router-Id that is being requested. This MUST NOT consist of all zeroes or all ones.

Prefix The prefix being requested. This field's size is Plen/8 rounded upwards.

A Seqno Request TLV prompts the receiving node to send a finite-metric Update for the prefix specified by the AE, Plen, and Prefix fields, with either a router-id different from what is specified by the Router-Id field, or a Seqno no less (modulo 2^16) than what is specified by the Seqno field. If this request cannot be satisfied

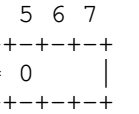
locally, then it is forwarded according to the rules set out in Section 3.8.1.2.

While a Seqno Request MAY be sent to a multicast address, it MUST NOT be forwarded to a multicast address and MUST NOT be forwarded to more than one neighbour. A request MUST NOT be forwarded if its Hop Count field is 1.

This TLV is self-terminating, and allows sub-TLVs.

4.7. Details of specific sub-TLVs

4.7.1. Pad1

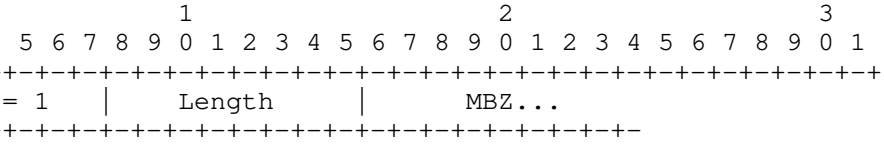


Fields :

Type Set to 0 to indicate a Pad1 sub-TLV.

This sub-TLV is silently ignored on reception. It is allowed within any TLV that allows sub-TLVs.

4.7.2. PadN



Fields :

Type Set to 1 to indicate a PadN sub-TLV.

Length The length of the body in octets, exclusive of the Type and Length fields.

MBZ Must be zero, set to 0 on transmission.

This sub-TLV is silently ignored on reception. It is allowed within any TLV that allows sub-TLVs.

5. IANA Considerations

IANA has registered the UDP port number 6696, called "babel", for use by the Babel protocol.

IANA has registered the IPv6 multicast group ff02::1:6 and the IPv4 multicast group 224.0.0.111 for use by the Babel protocol.

IANA has created a registry called "Babel TLV Types". The allocation policy for this registry is Specification Required [RFC8126] for Types 0-223, and Experimental Use for Types 224-254. The values in this registry are as follows:

Type	Name	Reference
0	Pad1	this document
1	PadN	this document
2	Acknowledgment Request	this document
3	Acknowledgment	this document
4	Hello	this document
5	IHU	this document
6	Router-Id	this document
7	Next Hop	this document
8	Update	this document
9	Route Request	this document
10	Seqno Request	this document
11	TS/PC	[RFC7298]
12	HMAC	[RFC7298]
13	Source-specific Update	[BABEL-SS]
14	Source-specific Request	[BABEL-SS]
15	Source-specific Seqno Request	[BABEL-SS]

16	MAC	[BABEL-MAC]
17	PC	[BABEL-MAC]
18	Challenge Request	[BABEL-MAC]
19	Challenge Reply	[BABEL-MAC]
20-223	Unassigned	
224-254	Reserved for Experimental Use	this document
255	Reserved for expansion of the type space	this document

IANA has created a registry called "Babel sub-TLV Types". The allocation policy for this registry is Specification Required for Types 0-111 and 128-239, and Experimental Use for Types 112-126 and 240-254. The values in this registry are as follows:

Type	Name	Reference
0	Pad1	this document
1	PadN	this document
2	Diversity	[BABEL-DIVERSITY]
3	Timestamp	[BABEL-RTT]
4-111	Unassigned	
112-126	Reserved for Experimental Use	this document
127	Reserved for expansion of the type space	this document
128	Source Prefix	[BABEL-SS]
129-239	Unassigned	
240-254	Reserved for Experimental Use	this document
255	Reserved for expansion of the type space	this document

IANA is instructed to create a registry called "Babel Address Encodings". The allocation policy for this registry is Specification Required for Address Encodings (AEs) 0-223, and Experimental Use for AEs 224-254. The values in this registry are as follows:

AE	Name	Reference
0	Wildcard address	this document
1	IPv4 address	this document
2	IPv6 address	this document
3	Link-local IPv6 address	this document
4-223	Unassigned	
224-254	Reserved for Experimental Use	this document
255	Reserved for expansion of the AE space	this document

IANA has created a registry called "Babel Flags Values". The allocation policy for this registry is Specification Required. IANA is instructed to rename this registry to "Babel Update Flags Values". The values in this registry are as follows:

Bit	Name	Reference
0	Default prefix	this document
1	Default Router-ID	this document
2-7	Unassigned	

IANA is instructed to create a new registry called "Babel Hello Flags Values". The allocation policy for this registry is Specification Required. The initial values in this registry are as follows:

Bit	Name	Reference
0	Unicast	this document
1-15	Unassigned	

IANA is instructed to replace all references to RFCs 6126 and 7557 in all of the registries mentioned above by references to this document.

6. Security Considerations

As defined in this document, Babel is a completely insecure protocol. Without additional security mechanisms, Babel trusts any information it receives in plaintext UDP datagrams and acts on it. An attacker that is present on the local network can impact Babel operation in a variety of ways; for example they can:

- spoof a Babel packet, and redirect traffic by announcing a route with a smaller metric, a larger sequence number, or a longer prefix;
- spoof a malformed packet, which could cause an insufficiently robust implementation to crash or interfere with the rest of the network;
- replay a previously captured Babel packet, which could cause traffic to be redirected, blackholed or otherwise interfere with the network.

When carried over IPv6, Babel packets are ignored unless they are sent from a link-local IPv6 address; since routers don't forward link-local IPv6 packets, this mitigates the attacks outlined above by restricting them to on-link attackers. No such natural protection exists when Babel packets are carried over IPv4, which is one of the reasons why it is recommended to deploy Babel over IPv6 (Section 3.1).

It is usually difficult to ensure that packets arriving at a Babel node are trusted, even in the case where the local link is believed to be secure. For that reason, it is RECOMMENDED that all Babel traffic be protected by an application-layer cryptographic protocol. There are currently two suitable mechanisms, which implement different tradeoffs between implementation simplicity and security:

- Babel over DTLS [BABEL-DTLS] runs the majority of Babel traffic over DTLS, and leverages DTLS to authenticate nodes and provide confidentiality and integrity protection;

- MAC authentication [BABEL-MAC] appends a message authentication code (MAC) to every Babel packet to prove that it originated at a node that knows a shared secret, and includes sufficient additional information to prove that the packet is fresh (not replayed).

Both mechanisms enable nodes to ignore packets generated by attackers without the proper credentials. They also ensure integrity of messages and prevent message replay. While Babel-DTLS supports asymmetric keying and ensures confidentiality, Babel-MAC has a much more limited scope (see Sections 1.1, 1.2 and 7 of [BABEL-MAC]). Since Babel-MAC is simpler and more lightweight, it is recommended in preference to Babel-DTLS in deployments where its limitations are acceptable, i.e., when symmetric keying is sufficient and where the routing information is not considered confidential.

Every implementation of Babel SHOULD implement BABEL-MAC.

One should be aware that the information that a mobile Babel node announces to the whole routing domain is sufficient to determine the mobile node's physical location with reasonable precision, which might cause privacy concerns even if the control traffic is protected from unauthenticated attackers by a cryptographic mechanism such as Babel-DTLS. This issue may be mitigated somewhat by using randomly chosen router-ids and randomly chosen IP addresses, and changing them often enough.

7. Acknowledgments

A number of people have contributed text and ideas to this specification. The authors are particularly indebted to Matthieu Boutier, Gwendoline Chouasne, Margaret Cullen, Donald Eastlake, Toke Hoiland-Jorgensen, Benjamin Kaduk, Joao Sobrinho and Martin Vigoureux. Earlier versions of this document greatly benefited from the input of Joel Halpern. The address compression technique was inspired by [PACKETBB].

8. References

8.1. Normative References

[BABEL-MAC]
Do, C., Kolodziejak, W., and J. Chroboczek, "MAC authentication for the Babel routing protocol", Internet Draft draft-ietf-babel-hmac-10, August 2019.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997.

[RFC793] Postel, J., "Transmission Control Protocol", RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, June 2017.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017.

8.2. Informative References

[BABEL-DIVERSITY]
Chroboczek, J., "Diversity Routing for the Babel Routing Protocol", draft-chroboczek-babel-diversity-routing-01 (work in progress), February 2016.

[BABEL-DTLS]
Decimo, A., Schinazi, D., and J. Chroboczek, "Babel Routing Protocol over Datagram Transport Layer Security", Internet Draft draft-ietf-babel-dtls-10, June 2020.

[BABEL-RTT]
Jonglez, B. and J. Chroboczek, "Delay-based Metric Extension for the Babel Routing Protocol", draft-ietf-babel-rtt-extension-00 (work in progress), April 2019.

[BABEL-SS]
Boutier, M. and J. Chroboczek, "Source-Specific Routing in Babel", draft-ietf-babel-source-specific-05 (work in progress), April 2019.

[DSDV] Perkins, C. and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers", ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications 234-244, 1994.

[DUAL] Garcia Luna Aceves, J., "Loop-Free Routing Using Diffusing Computations", IEEE/ACM Transactions on Networking 1:1, February 1993.

[EIGRP] Albrightson, B., Garcia Luna Aceves, J., and J. Boyle, "EIGRP -- a Fast Routing Protocol Based on Distance Vectors", Proc. Interop 94, 1994.

[ETX] De Couto, D., Aguayo, D., Bicket, J., and R. Morris, "A high-throughput path metric for multi-hop wireless networks", Proc. MobiCom 2003, 2003.

[IEEE802.11] IEEE, "IEEE Standard for Information technology--Telecommunications and information exchange between systems Local and metropolitan area networks--Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications", IEEE 802.11-2012, DOI 10.1109/ieeestd.2012.6178212, April 2012.

[IEN137] Cohen, D., "On holy wars and a plea for peace", IEN 137, April 1980.

[IS-IS] Standardization, I. O. F., "Information technology -- Telecommunications and information exchange between systems -- Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)", ISO/IEC 10589:2002, 2002.

[JITTER] Floyd, S. and V. Jacobson, "The synchronization of periodic routing messages", IEEE/ACM Transactions on Networking 2, 2, 122-136, April 1994.

[OSPF] Moy, J., "OSPF Version 2", RFC 2328, April 1998.

[PACKETBB] Clausen, T., Dearlove, C., Dean, J., and C. Adjih, "Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format", RFC 5444, February 2009.

[RFC2675] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, DOI 10.17487/RFC2675, August 1999.

[RFC3561] Perkins, C., Belding-Royer, E., and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing", RFC 3561, DOI 10.17487/RFC3561, July 2003, <https://www.rfc-editor.org/info/rfc3561>.

[RFC6126] Chroboczek, J., "The Babel Routing Protocol", RFC 6126, DOI 10.17487/RFC6126, April 2011.

[RFC7298] Ovsienko, D., "Babel Hashed Message Authentication Code (HMAC) Cryptographic Authentication", RFC 7298, DOI 10.17487/RFC7298, July 2014.

[RFC7557] Chroboczek, J., "Extension Mechanism for the Babel Routing Protocol", RFC 7557, DOI 10.17487/RFC7557, May 2015.

[RIP] Malkin, G., "RIP Version 2", RFC 2453, November 1998.

Appendix A. Cost and Metric Computation

The strategy for computing link costs and route metrics is a local matter; Babel itself only requires that it comply with the conditions given in Section 3.4.3 and Section 3.5.2. Different nodes may use different strategies in a single network and may use different strategies on different interface types. This section describes a set of strategies that have been found to work well in actual networks.

In summary, a node maintains per-neighbour statistics about the last 16 received Hello TLVs of each kind (Appendix A.1), it computes costs by using the 2-out-of-3 strategy (Appendix A.2.1) on wired links, and ETX (Appendix A.2.2) on wireless links. It uses an additive algebra for metric computation (Section 3.5.2).

A.1. Maintaining Hello History

For each neighbour, a node maintains two sets of Hello history, one for each kind of Hello, and an expected sequence number, one for Multicast and one for Unicast Hellos. Each Hello history is a vector of 16 bits, where a 1 value represents a received Hello, and a 0 value a missed Hello. For each kind of Hello, the expected sequence number, written ne, is the sequence number that is expected to be carried by the next received Hello from this neighbour.

Whenever it receives a Hello packet of a given kind from a neighbour, a node compares the received sequence number nr for that kind of Hello with its expected sequence number ne. Depending on the outcome of this comparison, one of the following actions is taken:

- if the two differ by more than 16 (modulo 2^16), then the sending node has probably rebooted and lost its sequence number; the whole associated neighbour table entry is flushed and a new one is created;

- otherwise, if the received nr is smaller (modulo 2^16) than the expected sequence number ne, then the sending node has increased its Hello interval without us noticing; the receiving node removes the last (ne - nr) entries from this neighbour's Hello history (we "undo history");

- otherwise, if nr is larger (modulo 2^16) than ne, then the sending node has decreased its Hello interval, and some Hellos were lost; the receiving node adds (nr - ne) 0 bits to the Hello history (we "fast-forward").

The receiving node then appends a 1 bit to the Hello history and sets ne to (nr + 1). If the Interval field of the received Hello is not zero, it resets the neighbour's hello timer to 1.5 times the advertised Interval (the extra margin allows for delay due to jitter).

Whenever either Hello timer associated to a neighbour expires, the local node adds a 0 bit to the corresponding Hello history, and increments the expected Hello number. If both Hello histories are empty (they contain 0 bits only), the neighbour entry is flushed; otherwise, the relevant hello timer is reset to the value advertised in the last Hello of that kind received from this neighbour (no extra margin is necessary in this case, since jitter was already taken into account when computing the timeout that has just expired).

A.2. Cost Computation

This section describes two algorithms suitable for computing costs (Section 3.4.3) based on Hello history. Appendix A.2.1 applies to wired links, and Appendix A.2.2 to wireless links. RECOMMENDED default values of the parameters that appear in these algorithms are given in Appendix B.

A.2.1. k-out-of-j

K-out-of-j link sensing is suitable for wired links that are either up, in which case they only occasionally drop a packet, or down, in which case they drop all packets.

The k-out-of-j strategy is parameterised by two small integers k and j, such that 0 < k <= j, and the nominal link cost, a constant C >= 1. A node keeps a history of the last j hellos; if k or more of

those have been correctly received, the link is assumed to be up, and the rxcost is set to C; otherwise, the link is assumed to be down, and the rxcost is set to infinity.

Since Babel supports two kinds of Hellos, a Babel node performs k-out-of-j twice for each neighbour, once on the Unicast and once on the Multicast Hello history. If either of the instances of k-out-of-j indicates that the link is up, then the link is assumed to be up, and the rxcost is set to C; if both instances indicate that the link is down, then the link is assumed to be down, and the rxcost is set to infinity. In other words, the resulting rxcost is the minimum of the rxcosts yielded by the two instances of k-out-of-j link sensing.

The cost of a link performing k-out-of-j link sensing is defined as follows:

- cost = FFFF hexadecimal if rxcost = FFFF hexadecimal;
- cost = txcost otherwise.

A.2.2. ETX

Unlike wired links which are bimodal (either up or down), wireless links exhibit continuous variation of the link quality. Naive application of hop-count routing in networks that use wireless links for transit tends to select long, lossy links in preference to shorter, lossless links, which can dramatically reduce throughput. For that reason, a routing protocol designed to support wireless links must perform some form of link-quality estimation.

The Expected Transmission Cost algorithm, or ETX [ETX], is a simple link-quality estimation algorithm that is designed to work well with the IEEE 802.11 MAC [IEEE802.11]. By default, the IEEE 802.11 MAC performs Automatic Repeat Query (ARQ) and rate adaptation on unicast frames, but not on multicast frames, which are sent at a fixed rate with no ARQ; therefore, measuring the loss rate of multicast frames yields a useful estimate of a link's quality.

A node performing ETX link quality estimation uses a neighbour's Multicast Hello history to compute an estimate, written beta, of the probability that a Hello TLV is successfully received. Beta can be computed as the fraction of 1 bits within a small number (say, 6) of the most recent entries in the Multicast Hello history, or it can be an exponential average, or some combination of both approaches. Let rxcost be 256 / beta.

Let alpha be MIN(1, 256/txcost), an estimate of the probability of successfully sending a Hello TLV. The cost is then computed by

cost = 256/(alpha * beta)

or, equivalently,

cost = (MAX(txcost, 256) * rxcost) / 256.

Since the IEEE 802.11 MAC performs ARQ on unicast frames, unicast frames do not provide a useful measure of link quality, and therefore ETX ignores the Unicast Hello history. Thus, a node performing ETX link-quality estimation will not route through neighbouring nodes unless they send periodic Multicast Hellos (possibly in addition to Unicast Hellos).

A.3. Route selection and hysteresis

Route selection (Section 3.6) is the process by which a node selects a single route among the routes that it has available towards a given destination. With Babel, any route selection procedure that only ever chooses feasible routes with a finite metric will yield a set of loop-free routes; however, in the presence of continuously variable metrics such as ETX (Appendix A.2.2), a naive route selection procedure might lead to persistent oscillations. Such oscillations can be limited or avoided altogether by implementing hysteresis within the route selection algorithm, i.e., by making the route selection algorithm sensitive to a route's history. Any reasonable hysteresis algorithm should yield good results; the following algorithm is simple to implement and has been successfully deployed in a variety of environments.

For every route R, in addition to the route's metric m(R), maintain a smoothed version of m(R) written ms(R) (we RECOMMEND computing ms(R) as an exponentially smoothed average (see Section 3.7 of [RFC793]) of m(R) with a time constant equal to the Hello interval multiplied by a small number such as 3). If no route to a given destination is selected, then select the route with the smallest metric, ignoring the smoothed metric. If a route R is selected, then switch to a route R' only when both m(R') < m(R) and ms(R') < ms(R).

Intuitively, the smoothed metric is a long-term estimate of the quality of a route. The algorithm above works by only switching routes when both the instantaneous and the long-term estimate of the route's quality indicate that switching is profitable.

Appendix B. Protocol parameters

The choice of time constants is a trade-off between fast detection of mobility events and protocol overhead. Two instances of Babel running with different time constants will interoperate, although the resulting worst-case convergence time will be dictated by the slower of the two.

The Hello interval is the most important time constant: an outage or a mobility event is detected within 1.5 to 3.5 Hello intervals. Due to Babel's use of a redundant route table, and due to its reliance on triggered updates and explicit requests, the Update interval has little influence on the time needed to reconverge after an outage: in practice, it only has a significant effect on the time needed to acquire new routes after a mobility event. While the protocol allows intervals as low as 10ms, such low values would cause significant amounts of protocol traffic for little practical benefit.

The following values have been found to work well in a variety of environments, and are therefore RECOMMENDED default values:

Multicast Hello Interval: 4 seconds.

Unicast Hello Interval: infinite (no Unicast Hellos are sent).

Link cost: estimated using ETX on wireless links; 2-out-of-3 with C=96 on wired links.

IHU Interval: the advertised IHU interval is always 3 times the Multicast Hello interval. IHUs are actually sent with each Hello on lossy links (as determined from the Hello history), but only with every third Multicast Hello on lossless links.

Update Interval: 4 times the Multicast Hello interval.

IHU Hold Time: 3.5 times the advertised IHU interval.

Route Expiry Time: 3.5 times the advertised update interval.

Request timeout: initially 2 seconds, doubled every time a request is resent, up to a maximum of three times.

Urgent timeout: 0.2 seconds.

Source GC time: 3 minutes.

Appendix C. Route filtering

Route filtering is a procedure where an instance of a routing protocol either discards some of the routes announced by its neighbours, or learns them with a metric that is higher than what would be expected. Like all distance-vector protocols, Babel has the ability to apply arbitrary filtering to the routes it learns, and implementations of Babel that apply different sets of filtering rules will interoperate without causing routing loops. The protocol's ability to perform route filtering is a consequence of the latitude given in Section 3.5.2: Babel can use any metric that is strictly monotonic, including one that assigns an infinite metric to a selected subset of routes. (See also Section 3.8.1, where requests for nonexistent routes are treated in the same way as requests for routes with infinite metric.)

It is not in general correct to learn a route with a metric smaller than the one it was announced with, or to replace a route's destination prefix with a more specific (longer) one. Doing either of these may cause persistent routing loops.

Route filtering is a useful tool, since it allows fine-grained tuning of the routing decisions made by the routing protocol. Accordingly, some implementations of Babel implement a rich configuration language that allows applying filtering to sets of routes defined, for example, by incoming interface and destination prefix.

In order to limit the consequences of misconfiguration, Babel implementations provide a reasonable set of default filtering rules even when they don't allow configuration of filtering by the user. At a minimum, they discard routes with a destination prefix in fe80::/64, ff00::/8, 127.0.0.1/32, 0.0.0.0/32 and 224.0.0.0/8.

Appendix D. Considerations for protocol extensions

Babel is an extensible protocol, and this document defines a number of mechanisms that can be used to extend the protocol in a backwards compatible manner:

- increasing the version number in the packet header;
- defining new TLVs;
- defining new sub-TLVs (with or without the mandatory bit set);
- defining new AEs;
- using the packet trailer.

This appendix is intended to guide designers of protocol extensions in choosing a particular encoding.

The version number in the Babel header should only be increased if the new version is not backwards compatible with the original protocol.

In many cases, an extension could be implemented either by defining a new TLV, or by adding a new sub-TLV to an existing TLV. For example, an extension whose purpose is to attach additional data to route updates can be implemented either by creating a new "enriched" Update TLV, by adding a non-mandatory sub-TLV to the Update TLV, or by adding a mandatory sub-TLV.

The various encodings are treated differently by implementations that do not understand the extension. In the case of a new TLV or of a sub-TLV with the mandatory bit set, the whole TLV is ignored by implementations that do not implement the extension, while in the case of a non-mandatory sub-TLV, the TLV is parsed and acted upon, and only the unknown sub-TLV is silently ignored. Therefore, a non-mandatory sub-TLV should be used by extensions that extend the Update in a compatible manner (the extension data may be silently ignored), while a mandatory sub-TLV or a new TLV must be used by extensions that make incompatible extensions to the meaning of the TLV (the whole TLV must be thrown away if the extension data is not understood).

Experience shows that the need for additional data tends to crop up in the most unexpected places. Hence, it is recommended that extensions that define new TLVs should make them self-terminating, and allow attaching sub-TLVs to them.

Adding a new AE is essentially equivalent to adding a new TLV: Update TLVs with an unknown AE are ignored, just like unknown TLVs. However, adding a new AE is more involved than adding a new TLV, since it creates a new set of compression state. Additionally, since the Next Hop TLV creates state specific to a given address family, as opposed to a given AE, a new AE for a previously defined address family must not be used in the Next Hop TLV if backwards compatibility is required. A similar issue arises with Update TLVs with unknown AEs establishing a new router-id (due to the Router-Id flag being set). Therefore, defining new AEs must be done with care if compatibility with unextended implementations is required.

The packet trailer is intended to carry cryptographic signatures that only cover the packet body; storing the cryptographic signatures in the packet trailer avoids clearing the signature before computing a hash of the packet body, and makes it possible to check a

cryptographic signature before running the full, stateful TLV parser. Hence, only TLVs that don't need to be protected by cryptographic security protocols should be allowed in the packet trailer. Any such TLVs should be easy to parse, and in particular should not require stateful parsing.

Appendix E. Stub Implementations

Babel is a fairly economic protocol. Updates take between 12 and 40 octets per destination, depending on the address family and how successful compression is; in a double-stack flat network, an average of less than 24 octets per update is typical. The route table occupies about 35 octets per IPv6 entry. To put these values into perspective, a single full-size Ethernet frame can carry some 65 route updates, and a megabyte of memory can contain a 20000-entry route table and the associated source table.

Babel is also a reasonably simple protocol. One complete implementation consists of less than 12 000 lines of C code, and it compiles to less than 120 kB of text on a 32-bit CISC architecture; about half of this figure is due to protocol extensions and user-interface code.

Nonetheless, in some very constrained environments, such as PDAs, microwave ovens, or abacuses, it may be desirable to have subset implementations of the protocol.

There are many different definitions of a stub router, but for the needs of this section a stub implementation of Babel is one that announces one or more directly attached prefixes into a Babel network but doesn't reannounce any routes that it has learnt from its neighbours, and always prefers the direct route to a directly attached prefix to a route learned over the Babel protocol, even when the prefixes are the same. It may either maintain a full routing table, or simply select a default gateway through any one of its neighbours that announces a default route. Since a stub implementation never forwards packets except from or to a directly attached link, it cannot possibly participate in a routing loop, and hence it need not evaluate the feasibility condition or maintain a source table.

No matter how primitive, a stub implementation must parse sub-TLVs attached to any TLVs that it understands and check the mandatory bit. It must answer acknowledgment requests and must participate in the Hello/IHU protocol. It must also be able to reply to seqno requests for routes that it announces and, and it should be able to reply to route requests.

Experience shows that an IPv6-only stub implementation of Babel can be written in less than 1000 lines of C code and compile to 13 kB of text on 32-bit CISC architecture.

Appendix F. Compatibility with previous versions

The protocol defined in this document is a successor to the protocol defined in [RFC6126] and [RFC7557]. While the two protocols are not entirely compatible, the new protocol has been designed so that it can be deployed in existing RFC 6126 networks without requiring a flag day.

There are three optional features that make this protocol incompatible with its predecessor. First of all, RFC 6126 did not define Unicast hellos (Section 3.4.1), and an implementation of RFC 6126 will mis-interpret a Unicast Hello for a Multicast one; since the sequence number space of Unicast Hellos is distinct from the sequence space of Multicast Hellos, sending a Unicast Hello to an implementation of RFC 6126 will confuse its link quality estimator. Second, RFC 6126 did not define unscheduled Hellos, and an implementation of RFC 6126 will mis-parse Hellos with an interval equal to 0. Finally, RFC 7557 did not define mandatory sub-TLVs (Section 4.4), and thus, an implementation of RFCs 6126 and 7557 will not correctly ignore a TLV that carries an unknown mandatory sub-TLV; depending on the sub-TLV, this might cause routing pathologies.

An implementation of this specification that never sends Unicast or unscheduled Hellos and doesn't implement any extensions that use mandatory sub-TLVs is safe to deploy in a network in which some nodes implement the protocol described in RFCs 6126 and 7557.

Two changes need to be made to an implementation of RFCs 6126 and 7557 so that it can safely interoperate in all cases with implementations of this protocol. First, it needs to be modified to either ignore or process Unicast and unscheduled Hellos. Second, it needs to be modified to parse sub-TLVs of all the TLVs that it understands and that allow sub-TLVs, and to ignore the TLV if an unknown mandatory sub-TLV is found. It is not necessary to parse unknown TLVs, as these are ignored in any case.

There are other changes, but these are not of a nature to prevent interoperability:

- the conditions on route acquisition (Section 3.5.3) have been relaxed;
- route selection should no longer use the route's sequence number (Section 3.6);

- the format of the packet trailer has been defined (Section 4.2);
- router-ids with a value of all-zeros or all-ones have been forbidden (Section 4.1.3);
- the compression state is now specific to an address family rather than an address encoding (Section 4.5);
- packet pacing is now recommended (Section 3.1).

Appendix G. Changes from previous versions

[RFC Editor: Please delete this section before publication.]

G.1. Changes since RFC 6126

- Changed UDP port number to 6696.
- Consistently use router-id rather than id.
- Clarified that the source garbage collection timer is reset after sending an update even if the entry was not modified.
- In section "Seqno Requests", fixed an erroneous "route request".
- In the description of the Seqno Request TLV, added the description of the Router-Id field.
- Made router-ids all-0 and all-1 forbidden.

G.2. Changes since draft-ietf-babel-rfc6126bis-00

- Added security considerations.

G.3. Changes since draft-ietf-babel-rfc6126bis-01

- Integrated the format of sub-TLVs.
- Mentioned for each TLV whether it supports sub-TLVs.
- Added Appendix D.
- Added a mandatory bit in sub-TLVs.
- Changed compression state to be per-AF rather than per-AE.
- Added implementation hint for the routing table.

- Clarified how router-ids are computed when bit 0x40 is set in Updates.
- Relaxed the conditions for sending requests, and tightened the conditions for forwarding requests.
- Clarified that neighbours should be acquired at some point, but it doesn't matter when.

G.4. Changes since draft-ietf-babel-rfc6126bis-02

- Added Unicast Hellos.
- Added unscheduled (interval-less) Hellos.
- Changed Appendix A to consider Unicast and unscheduled Hellos.
- Changed Appendix B to agree with the reference implementation.
- Added optional algorithm to avoid the hold time.
- Changed the table of pending seqno requests to be indexed by router-id in addition to prefixes.
- Relaxed the route acquisition algorithm.
- Replaced minimal implementations by stub implementations.
- Added acknowledgments section.

G.5. Changes since draft-ietf-babel-rfc6126bis-03

- Clarified that all the data structures are conceptual.
- Made sending and receiving Multicast Hellos a SHOULD, avoids expressing any opinion about Unicast Hellos.
- Removed opinion about Multicast vs. Unicast Hellos (Appendix A.4).
- Made hold-time into a SHOULD rather than MUST.
- Clarified that Seqno Requests are for a finite-metric Update.
- Clarified that sub-TLVs Pad1 and PadN are allowed within any TLV that allows sub-TLVs.
- Updated IANA Considerations.

- Updated Security Considerations.
- Renamed routing table back to route table.
- Made buffering outgoing updates a SHOULD.
- Weakened advice to use modified EUI-64 in router-ids.
- Added information about sending requests to Appendix B.
- A number of minor wording changes and clarifications.

G.6. Changes since draft-ietf-babel-rfc6126bis-03

Minor editorial changes.

G.7. Changes since draft-ietf-babel-rfc6126bis-04

- Renamed isotonicity to left-distributivity.
- Minor clarifications to unicast hellos.
- Updated requirements boilerplate to RFC 8174.
- Minor editorial changes.

G.8. Changes since draft-ietf-babel-rfc6126bis-05

- Added information about the packet trailer, now that it is used by draft-ietf-babel-hmac.

G.9. Changes since draft-ietf-babel-rfc6126bis-06

- Added references to security documents.

G.10. Changes since draft-ietf-babel-rfc6126bis-07

- Added list of obsoleted drafts to the abstract.
- Updated references.

G.11. Changes since draft-ietf-babel-rfc6126bis-08

- Added recommendation that route selection should not take seqnos into account.

G.12. Changes since draft-ietf-babel-rfc6126bis-09

- Editorial changes only.

G.13. Changes since draft-ietf-babel-rfc6126bis-10

- Editorial changes only.

G.14. Changes since draft-ietf-babel-rfc6126bis-11

- Added recommendation that control traffic should be carried over IPv6 only.

G.15. Changes since draft-ietf-babel-rfc6126bis-12

- Removed appendix about software availability.
- Expanded appendix about recommended values and added more references to it in the body of the document.
- Added appendix about route filtering.
- Clarified definition of mandatory bit.
- Added recommendations for packet pacing.
- Made time limiting of full updates a SHOULD.
- Normative language in a few more places.
- Removed normative language from stub implementations.
- Added requirement to clear the undefined bits in an Update.
- Added error checking requirements.
- Reworked security considerations.
- Added "in octets" and "in bits" in random places.
- Inserted full IANA registries.
- Editorial changes.

G.16. Changes since draft-ietf-babel-rfc6126bis-13

- Added a section about compatibility with 6126.
- Added AE registry to IANA considerations.
- Replaced Babel-HMAC with Babel-MAC, consistent with the change in draft-ietf-babel-hmac.
- Removed section about external sources of willingness; filtering is a better approach.
- Added recommendation to use a cost of 96 on wired links.
- Editorial changes.

G.17. Changes since draft-ietf-babel-rfc6126bis-14

- Added unscheduled Hellos to compatibility considerations.
- Created new appendix about route selection.
- Reworked security considerations.
- Added some comments about packet pacing and low update intervals.

G.18. Changes since draft-ietf-babel-rfc6126bis-15

- Implementing Babel-MAC is now recommended.

G.19. Changes since draft-ietf-babel-rfc6126bis-16

- Make the values in Appendix B normatively recommended defaults.

G.20. Changes since draft-ietf-babel-rfc6126bis-17

- Hysteresis in route selection is now RECOMMENDED.
- Additive metric algebra is now RECOMMENDED default.
- 2-out-of-3 cost computation is now RECOMMENDED on LANs.
- Reference to RFC 793 Section 3.7 added as exponential smoothing example.

G.21. Changes since draft-ietf-babel-rfc6126bis-18

- Reserved Address Encodings 224-254 for Experimental Use, and 255 for future expansion.

G.22. Changes since draft-ietf-babel-rfc6126bis-19

- Mention that multi-octet fields are in big-endian.

- Minor typos and clarifications.

Authors' Addresses

Juliusz Chroboczek
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: jch@irif.fr

David Schinazi
Google LLC
1600 Amphitheatre Parkway
Mountain View, California 94043
USA

Email: dschinazi.ietf@gmail.com

Babel Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 24, 2019

M. Jethanandani
VMware
B. Stark
AT&T

October 21, 2018

YANG Data Model for Babel
draft-mahesh-babel-yang-model-00

Abstract

This document defines a data model for the Babel routing protocol. The data model is defined using the YANG data modeling language.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119][RFC8174] when, and only when, they appear in all capitals, as shown here..

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 24, 2019.

Copyright Notice

Table of Contents

1. Introduction

This document defines a data model for the Babel routing protocol [I-D.ietf-babel-rfc6126bis]. The data model is defined using the YANG [RFC7950] data modeling language. It is based on the Babel Information Model [I-D.ietf-babel-information-model].

Artwork in this document contains shorthand references to drafts in progress. Please apply the following replacements

- "XXXX" --> the assigned RFC value for this draft both in this draft and in the YANG models under the revision statement.

- Revision date in model, in the format 2018-04-27 needs to get updated with the date the draft gets approved. The date also needs to get reflected on the line with <CODE BEGINS>.

1.1. Definitions and Acronyms

-

1.2. Tree Diagram

For a reference to the annotations used in tree diagrams included in this draft, please see YANG Tree Diagrams [RFC8340].

2. Babel Module

This document defines a YANG 1.1 [RFC7950] data model for the configuration and management of Babel. The YANG module is based on the Babel Information Model [I-D.ietf-babel-information-model].

2.1. Information Model

2.2. YANG Module

This module imports definitions from Common YANG Data Types [RFC6991].

```
module: ietf-babel
  +--rw babel!
     +--rw version?                  string
     +--rw enable?                   boolean
     +--rw router-id                 binary
     +--rw link-type*                identityref
     +--ro sequence-number?          yang:counter32
     +--rw cost-compute-algorithm*   identityref
     +--rw security-supported*       identityref
     +--rw transport
     |  +--rw udp-port?      inet:port-number
     |  +--rw mcast-group?   inet:ip-address
     +--rw interfaces* [reference]
     |  +--rw reference                if:interface-ref
     |  +--rw enable?                  boolean
     |  +--rw link-type?               identityref
     |  +--ro mcast-hello-seqno?       int16
     |  +--ro ucast-hello-seqno?       int16
     |  +--ro mcast-hello-interval?    int16
     |  +--ro ucast-hello-interval?    int16
     |  +--rw update-interval?         uint32
     |  +--rw external-cost?           uint32
     |  +--rw message-log-enable?      boolean
     |  +--rw message-log* [log-time]
     |  |  +--rw log-time     yang:timestamp
     |  |  +--rw log-entry?   string
     |  +--rw neighbor-objects* [neighbor-address]
     |  |  +--rw neighbor-address          inet:ip-address
     |  |  +--rw hello-mcast-history?      string
     |  |  +--rw hello-ucast-history?      string
```

```
|  |  +--rw txcost?                  int32
|  |  +--rw exp-mcast-hello-seqno?   int32
|  |  +--rw exp-ucast-hello-seqno?   int32
|  |  +--rw neighbor-ihu-interval?   int32
|  |  +--rw rxcost?                  int32
|  |  +--rw cost?                    int32
|  +--rw security* [mechanism]
|     +--rw mechanism               string
|     +--rw enable?                 boolean
|     +--rw self-cred* [id]
|     |  +--rw id      string
|     |  +--rw cred?   binary
|     +--rw trust* [id]
|     |  +--rw id      string
|     |  +--rw cred?   binary
|     +--rw credvalid-log-enable?   boolean
|     +--rw credvalid-log* [log-time]
|        +--rw log-time     yang:timestamp
|        +--rw log-entry?   string
+--rw routes* [prefix]
|  +--rw prefix                    inet:ip-address
|  +--rw prefix-length?            inet:ip-prefix
|  +--rw router-id?                binary
|  +--rw neighbor?
|  |       -> ../../interfaces/neighbor-objects/neighbor-address
|  +--rw (metric)
|  |  +--:(received-metric)
|  |  |  +--rw received-metric?    int32
|  |  +--:(calcuated-metric)
|  |     +--rw calcuated-metric?   int32
|  +--rw seqno?                    int32
|  +--rw next-hop?                 inet:ip-address
|  +--rw feasible?                 boolean
|  +--rw selected?                 boolean
+--rw security* [mechanism]
   +--rw mechanism               string
   +--rw enable?                 boolean
   +--rw self-cred* [id]
   |  +--rw id      string
   |  +--rw cred?   binary
   +--rw trust* [id]
   |  +--rw id      string
   |  +--rw cred?   binary
   +--rw credvalid-log-enable?   boolean
   +--rw credvalid-log* [log-time]
      +--rw log-time     yang:timestamp
      +--rw log-entry?   string
```

```
  <CODE BEGINS> file "ietf-babel@2018-10-21.yang"

  module ietf-babel {
    yang-version 1.1;
    namespace "urn:ietf:params:xml:ns:yang:ietf-babel";
    prefix babel;

    import ietf-yang-types {
      prefix yang;
      reference
        "RFC 6991 - Common YANG Data Types.";
    }
    import ietf-inet-types {
      prefix inet;
      reference
        "RFC 6991 - Common YANG Data Types.";
    }
    import ietf-interfaces {
      prefix if;
      reference
        "RFC 8343 - A YANG Data Model for Interface Management";
    }

    organization
      "IETF Babel routing protocl Working Group";

    contact
      "WG Web: http://tools.ietf.org/wg/babel/
       WG List: babel@ietf.org

       Editor: Mahesh Jethanandani
               mjethanandani@gmail.com
       Editor: Barbara Stark
               bs7652@att.com";

    description
      "This YANG module defines a model for the Babel routing
       protocol.

       Copyright (c) 2018 IETF Trust and the persons identified as
       the document authors.  All rights reserved.
       Redistribution and use in source and binary forms, with or
       without modification, is permitted pursuant to, and subject
       to the license terms contained in, the Simplified BSD
       License set forth in Section 4.c of the IETF Trust's Legal
       Provisions Relating to IETF Documents
       (http://trustee.ietf.org/license-info).
```

```
       This version of this YANG module is part of RFC XXXX; see
       the RFC itself for full legal notices.";

    revision 2018-10-21 {
      description
        "Initial version.";
      reference
        "RFC XXX: Babel YANG Data Model.";
    }

    /*
     * Identities
     */
    identity babel-link-type {
      description
        "Base identity from which all Babel Link Types are derived.";
    }

    identity ethernet {
      base "babel-link-type";
      description
        "Ethernet link type for Babel Routing Protocol.";
    }
    identity other {
      base "babel-link-type";
      description
        "Other link type for Babel Routing Protocol.";
    }
    identity tunnel {
      base "babel-link-type";
      description
        "Tunnel link type for Babel Routing Protocol.";
    }
    identity wireless {
      base "babel-link-type";
      description
        "Wireless link type for Babel Routing Protocol.";
    }
    identity moca {
      base "babel-link-type";
      description
        "Multimedia over Coax Alliance.";
    }
    identity g-hn-over-coax {
      base "babel-link-type";
      description
        "G.hn over coax.";
      reference
```

```
        "G.9960: Unified high-speed wireline-base home networking
         transceivers.";
    }
    identity g-hn-over-powerline {
      base "babel-link-type";
      description
        "G.hn over powerline.";
      reference
        "G.9960: Unified high-speed wireline-base home networking
         transceivers.";
    }
    identity home-plug {
      base "babel-link-type";
      description
        "HomePlug Power Alliance.";
      reference
        "IEEE 1901: HD-PC";
    }
    identity ieee-802-15 {
      base "babel-link-type";
      description
        "Wireless Personal Area Networks (WPAN).";
      reference
        "IEEE 802.15: Wireless Personal Area Networks (WPAN).";
    }

    identity babel-cost-compute-algorithm {
      description
        "Base identity from which all Babel cost compute algorithms
         are derived.";
    }
    identity k-out-of-j {
      base "babel-cost-compute-algorithm";
      description
        "k-out-of-j algorithm.";
    }
    identity etx {
      base "babel-cost-compute-algorithm";
      description
        "Expected Transmission Count.";
    }

    /*
     * Babel type identities
     */
    identity babel-security-supported {
      description
        "Base identity from which all Babel security types are
```

```
         derived.";
    }

    /*
     * Features
     */

    /*
     * Features supported
     */

    /*
     * Typedefs
     */
    typedef base64 {
      type string {
        pattern '(([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{3}=|'
              + '[A-Za-z0-9+/]{2}==)?){1}';
      }
      description
        "A binary-to-text encoding scheme to represent binary data in
         an ASCII string format.";
      reference
        "RFC 4648, The Base16, Base32, and Base64 Data Encodings";
    }

    /*
     * Groupings
     */
    grouping log {
      leaf log-time {
        type yang:timestamp;
        description
          "The date and time (according to the device internal
           clock setting, which may be a time relative to boot
           time, acquired from NTP, configured by the user, etc.)
           when this log entry was created.";
        reference
          "RFC YYYY, Babel Information Model, Section 4.2.";
      }

      leaf log-entry {
        type string;
        description
          "The logged message, as a string of utf-8 encoded hex
           characters.";
        reference
          "RFC YYYY, Babel Information Model, Section 4.2.";
```

```
      }
      description
        "A babel-log-obj list.";
      reference
        "RFC YYYY, Babel Information Model, Section 4.2.";
    }

    grouping credential {
      leaf id {
        type string;
        description
          "An identifier that identifies this credential uniquely.";
      }

      leaf cred {
        type binary;
        description
          "A credential, such as an X.509 certificate, a public key,
           etc. used for signing and/or encrypting babel messages.";
        reference
          "RFC YYYY, Babel Information Model, Section 4.1.";
      }
      description
        "A babel-credential-obj list.";
      reference
        "RFC YYYY, Babel Information Model, Section 4.1.";
    }

    grouping security {
      leaf mechanism {
        type string;
        description
          "The name of the security mechanism this object instance
           is about. The value MUST be the same as one of the
           identities listed as the babel-security-supported
           parameter.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.5.";
      }

      leaf enable {
        type boolean;
        description
          "If true, the security mechanism is running. If false,
           the security mechanism is not currently running.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.5.";
      }
```

```
      list self-cred {
        key "id";

        uses credential;
        description
          "Credentials this router presents to participate in the
           enabled security mechanism. Any private key component of
           a credential MUST NOT be readable. Adding and deleting
           credentials MAY be allowed.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.5.";
      }

      list trust {
        key "id";

        uses credential;
        description
          "A list of credential-obj objects that identify the
           credentials of routers whose babel messages may be
           trusted or of a certificate authority (CA) whose signing
           of a router's credentials implies the router credentials
           can be trusted, in the context of this security
           mechanism. How a security mechanism interacts with this
           list is determined by the mechanism. A security algorithm
           may do additional validation of credentials, such as
           checking validity dates or revocation lists, so presence
           in this list may not be sufficient to determine trust.
           Adding and deleting credentials MAY be allowed.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.5.";
      }

      leaf credvalid-log-enable {
        type boolean;
        description
          "If true, logging of messages that include credentials
           used for authentication is enabled. If false, these
           messages are not logged.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.5.";
      }

      list credvalid-log {
        key "log-time";

        uses log;
        description
```

```
          "Log entries that have the timestamp a message containing
           credentials used for peer authentication (e.g., DTLS
           Server Hello) was received on a Babel port, and the
           entire received message (including Ethernet frame and IP
           headers, if possible); an implementation must restrict
           the size of this log, but how and what size is
           implementation-specific.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.5.";
      }
      description
        "A babel-security-obj list.";
      reference
        "RFC YYYY, Babel Information Model, Section 3.5.";
    }

    /*
     * Data model
     */
    container babel {
      presence "A Babel container.";
      description
        "This is a top level container for the Babel routing protocol.";

      leaf version {
        type string;
        description
          "This is the version of the babel protocol implemented.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }

      leaf enable {
        type boolean;
        default false;
        description
          "When written, it configures whether the protocol should be
           enabled. A read from the <running> or <intended> datastore
           therefore indicates the configured adminstrative value of
           whether the protocol is enabled or not.

           A read from the <operational> datastore indicates whether
           the protocol is actually running or not, i.e. it indicates
           the operational state of the protocol.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }
```

```
      leaf router-id {
        type binary;
        mandatory "true";
        description
          "Every Babel speaker is assigned a router-id, which is an
           arbitrary string of 8 octets that is assumed to be unique
           across the routing domain";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1,
           rfc6126bis, The Babel Routing Protocol. Section 3.";
      }

      leaf-list link-type {
        type identityref {
          base "babel-link-type";
        }
        description
          "Link types supported by this implementation of Babel.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }

      leaf sequence-number {
        type yang:counter32;
        config false;
        description
          "Sequence number included in route updates for routes
           originated by this node.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }

      leaf-list cost-compute-algorithm {
        type identityref {
          base "babel-cost-compute-algorithm";
        }
        description
          "List of cost compute algorithms supported by this
           implementation of Babel.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }

      leaf-list security-supported {
        type identityref {
          base "babel-security-supported";
        }
        description
```

```
          "Babel security mechanism used by this implementation or
           per interface.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }

      container transport {
        leaf udp-port {
          type inet:port-number;
          default "6696";
          description
            "UDP port for sending and receiving Babel messages. The
             default port is 6696.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.2.";
        }

        leaf mcast-group {
          type inet:ip-address;
          default "ff02:0:0:0:0:0:1:6";
          description
            "Multicast group for sending and receiving multicast
             announcements on IPv6.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.2.";
        }
        description
          "Babel Transport object.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }
      list interfaces {
        key "reference";

        leaf reference {
          type if:interface-ref;
          description
            "Reference to an interface object as defined by the data
             model (e.g., YANG, BBF TR-181); data model is assumed to
             allow for referencing of interface objects which may be at
             any layer (physical, Ethernet MAC, IP, tunneled IP, etc.).
             Referencing syntax will be specific to the data model. If
             there is no set of interface objects available, this should
             be a string that indicates the interface name used by the
             underlying operating system.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }
```

```
        leaf enable {
          type boolean;
          default "true";
          description
            "If true, babel sends and receives messages on this
             interface. If false, babel messages received on this
             interface are ignored and none are sent.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        leaf link-type {
          type identityref {
            base babel-link-type;
          }
          description
            "Indicates the type of link. Set of values of supported
             link types where the following enumeration values MUST
             be supported when applicable: 'ethernet', 'wireless',
             'tunnel', and 'other'. Additional values MAY be
             supported.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        leaf mcast-hello-seqno {
          type int16;
          config false;
          description
            "The current sequence number in use for multicast hellos
             sent on this interface.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        leaf ucast-hello-seqno {
          type int16;
          config false;
          description
            "The current sequence number in use for unicast hellos
             sent on this interface.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        leaf mcast-hello-interval {
          type int16;
          config false;
          description
            "The current multicast hello interval in use for hellos
             sent on this interface.";
          reference
```

```
          "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        leaf ucast-hello-interval {
          type int16;
          config false;
          description
            "The current unicast hello interval in use for hellos sent
             on this interface.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        leaf update-interval {
          type uint32;
          description
            "The current update interval in use for this interface.";
          reference
        "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        leaf external-cost {
          type uint32;
          description
            "External input to cost of link of this interface. If
             supported, this is a value that is added to the metrics
             of routes learned over this interface. How an
             implementation uses the value is up to the implementation,
             which means the use may not be consistent across
             implementations.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        leaf message-log-enable {
          type boolean;
          description
            "If true, logging of babel messages received on this
             interface is enabled; if false, babel messages are not
             logged.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }

        list message-log {
          key "log-time";

          uses log;
          description
            "Log entries that have timestamp of a received Babel
             message and the entire received Babel message (including
             Ethernet frame and IP headers, if possible). An
```

```
            implementation must restrict the size of this log, but how
            and what size is implementation specific.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }

        list neighbor-objects {
          key "neighbor-address";

          leaf neighbor-address {
            type inet:ip-address;
            description
              "IPv4 or v6 address the neighbor sends messages from.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }

          leaf hello-mcast-history {
            type string;
            description
              "The multicast Hello history of whether or not the
               multicast Hello messages prior to babel-exp-mcast-
               hello-seqno were received, with a '1' for the most
               recent Hello placed in the most significant bit and
               prior Hellos shifted right (with '0' bits placed
               between prior Hellos and most recent Hello for any
               not-received Hellos); represented as a string using
               utf-8 encoded hex digits where a '1' bit = Hello
               received and a '0' bit = Hello not received.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }

          leaf hello-ucast-history {
            type string;
            description
              "The unicast Hello history of whether or not the
               unicast Hello messages prior to babel-exp-ucast-
               hello-seqno were received, with a '1' for the most
               recent Hello placed in the most significant bit and
               prior Hellos shifted right (with '0' bits placed
               between prior Hellos and most recent Hello for any
               not-received Hellos); represented as a string using
               utf-8 encoded hex digits where a '1' bit = Hello
               received and a '0' bit = Hello not received.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }
```

```
          leaf txcost {
            type int32;
            description
              "Transmission cost value from the last IHU packet
               received from this neighbor, or maximum value
               (infinity) to indicates the IHU hold timer for this
               neighbor has expired description.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }

          leaf exp-mcast-hello-seqno {
            type int32;
            description
              "Expected multicast Hello sequence number of next Hello
               to be received from this neighbor; if multicast Hello
               messages are not expected, or processing of multicast
               messages is not enabled, this MUST be 0.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }

          leaf exp-ucast-hello-seqno {
            type int32;
            description
              "Expected unicast Hello sequence number of next Hello to
               be received from this neighbor; if unicast Hello
               messages are not expected, or processing of unicast
               messages is not enabled, this MUST be 0.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }

          leaf neighbor-ihu-interval {
            type int32;
            description
              "Current IHU interval for this neighbor.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }

          leaf rxcost {
            type int32;
            description
              "Reception cost calculated for this neighbor. This value
               is usually derived from the Hello history, which may be
               combined with other data, such as statistics maintained
               by the link layer. The rxcost is sent to a neighbour in
```

```
              each IHU.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }

          leaf cost {
            type int32;
            description
              "Link cost is computed from the values maintained in
               the neighbour table. The statistics kept in the neighbour
               table about the reception of Hellos, and the txcost
               computed from received IHU packets.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.4.";
          }
          description
            "A set of Babel Neighbor Object.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }

        list security {
          key "mechanism";

          uses security;
          description
            "A security-obj object that applies to this interface. If
             implemented, this allows security to be enabled only on
             specific interfaces or allows different security mechanisms
             to be enabled on different interfaces.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.3.";
        }
        description
          "A set of Babel Interface objects.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }

      list routes {
        key "prefix";

        leaf prefix {
          type inet:ip-address;
          description
            "Prefix (expressed in IP address format) for which this
             route is advertised.";
          reference
```

```
          "RFC YYYY, Babel Information Model, Section 3.6.";
        }

        leaf prefix-length {
          type inet:ip-prefix;
          description
            "Length of the prefix for which this route is advertised.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.6.";
        }

        leaf router-id {
          type binary;
          description
            "router-id of the source router for which this route is
             advertised.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.6.";
        }

        leaf neighbor {
          type leafref {
            path "../../interfaces/neighbor-objects/neighbor-address";
          }
          description
            "Reference to the babel-neighbors entry for the neighbor
             that advertised this route.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.6.";
        }

        choice metric {
          mandatory "true";
          leaf received-metric {
            type int32;
            description
              "The metric with which this route was advertised by the
               neighbor, or maximum value (infinity) to indicate a the
               route was recently retracted and is temporarily
               unreachable. this metric will be 0 (zero) if the route
               was not received from a neighbor but was generated
               through other means. Either babel-route-calculated-metric
               or babel-route-received-metric MUST be provided.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.6,
               draft-ietf-babel-rfc6126bis, The Babel Routing Protocol,
               Section 3.5.5.";
          }
```

```
          leaf calcuated-metric {
            type int32;
            description
              "A calculated metric for this route. How the metric is
               calculated is implementation-specific. Maximum value
               (infinity) indicates the route was recently retracted
               and is temporarily unreachable. Either
               babel-route-calculated-metric or
               babel-route-received-metric MUST be provided.";
            reference
              "RFC YYYY, Babel Information Model, Section 3.6,
               draft-ietf-babel-rfc6126bis, The Babel Routing Protocol,
               Section 3.5.5.";
          }
          description
            "Either babel-route-calculated-metric or
             babel-route-received-metric MUST be provided.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.6,
             draft-ietf-babel-rfc6126bis, The Babel Routing Protocol,
             Section 3.5.5.";
        }

        leaf seqno {
          type int32;
          description
            "The sequence number with which this route was advertised.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.6.";
        }

        leaf next-hop {
          type inet:ip-address;
          description
            "The next-hop address of this route. This will be empty if
             this route has no next-hop address.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.6.";
        }

        leaf feasible {
          type boolean;
          description
            "A boolean flag indicating whether this route is feasible.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.6,
             draft-ietf-babel-rfc6126bis, The Babel Routing Protocol,
             Section 3.5.1.";
```

```
        }

        leaf selected {
          type boolean;
          description
            "A boolean flag indicating whether this route is selected,
             i.e., whether it is currently being used for forwarding and
             is being advertised.";
          reference
            "RFC YYYY, Babel Information Model, Section 3.6.";
        }
        description
          "A set of babel-route-obj objects. Includes received and
           routes routes.";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }

      list security {
        key "mechanism";

        uses security;
        description
          "A security-obj object that applies to all interfaces. If this
           object is implemented, it allows a security mechanism to be
           enabled or disabled in a manner that applies to all Babel
           messages on all interfaces";
        reference
          "RFC YYYY, Babel Information Model, Section 3.1.";
      }
    }
  }

  <CODE ENDS>
```

3. IANA Considerations

This document registers ?? URIs and ?? YANG modules.

3.1. URI Registrations

3.2. YANG Module Name Registration

This document registers ?? YANG module in the YANG Module Names registry YANG [RFC6020].

```
name:
namespace: urn:ietf:params:xml:ns:yang:
prefix: babel
reference: RFC XXXX
```

4. Security Considerations

The YANG module specified in this document defines a schema for data that is designed to be accessed via network management protocol such as NETCONF [RFC6241] or RESTCONF [RFC8040]. The lowest NETCONF layer is the secure transport layer and the mandatory-to-implement secure transport is SSH [RFC6242]. The lowest RESTCONF layer is HTTPS, and the mandatory-to-implement secure transport is TLS [RFC8446].

The NETCONF Access Control Model (NACM [RFC8341]) provides the means to restrict access for particular NETCONF users to a pre-configured subset of all available NETCONF protocol operations and content.

There are a number of data nodes defined in the YANG module which are writable/creatable/deletable (i.e., config true, which is the default). These data nodes may be considered sensitive or vulnerable in some network environments. Write operations (e.g., <edit-config>) to these data nodes without proper protection can have a negative effect on network operations.

These are the subtrees and data nodes and their sensitivity/ vulnerability:

5. Acknowledgements

6. References

6.1. Normative References

[I-D.ietf-babel-rfc6126bis]
Chroboczek, J. and D. Schinazi, "The Babel Routing Protocol", draft-ietf-babel-rfc6126bis-05 (work in progress), May 2018.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6991] Schoenwaelder, J., Ed., "Common YANG Data Types", RFC 6991, DOI 10.17487/RFC6991, July 2013, <https://www.rfc-editor.org/info/rfc6991>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

6.2. Informative References

[I-D.ietf-babel-information-model] Stark, B., "Babel Information Model", draft-ietf-babel-information-model-03 (work in progress), June 2018.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <https://www.rfc-editor.org/info/rfc6020>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/info/rfc6241>.

[RFC6242] Wasserman, M., "Using the NETCONF Protocol over Secure Shell (SSH)", RFC 6242, DOI 10.17487/RFC6242, June 2011, <https://www.rfc-editor.org/info/rfc6242>.

[RFC8040] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", RFC 8040, DOI 10.17487/RFC8040, January 2017, <https://www.rfc-editor.org/info/rfc8040>.

[RFC8340] Bjorklund, M. and L. Berger, Ed., "YANG Tree Diagrams", BCP 215, RFC 8340, DOI 10.17487/RFC8340, March 2018, <https://www.rfc-editor.org/info/rfc8340>.

[RFC8341] Bierman, A. and M. Bjorklund, "Network Configuration Access Control Model", STD 91, RFC 8341, DOI 10.17487/RFC8341, March 2018, <https://www.rfc-editor.org/info/rfc8341>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

Appendix A. An Appendix

Authors' Addresses

Mahesh Jethanandani
VMware
California
USA

Email: mjethanandani@gmail.com

Barbara Stark
AT&T
Atlanta, GA
USA

Email: barbara.stark@att.com