Stream:	Internet Engineering Task Force (IETF)
RFC:	8966
Obsoletes:	6126, 7557
Category:	Standards Track
Published:	January 2021
ISSN:	2070-1721
Authors:	J. Chroboczek	D. Schinazi
---	---	---
	IRIF, University of Paris-Diderot	*Google LLC*

RFC 8966
The Babel Routing Protocol

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 RFC 6126 and RFC 7557.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8966.

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

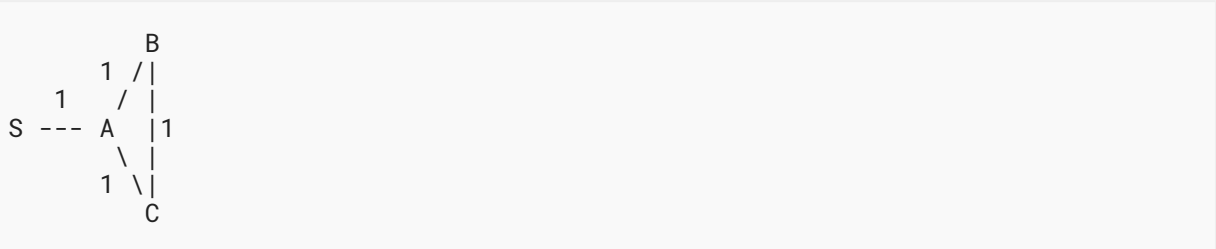
Every node A maintains two pieces of data: its estimated distance to S, written D(A), and its next-hop router to S, written NH(A). Initially, D(S) = 0, D(A) is infinite, and NH(A) is undefined.

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.

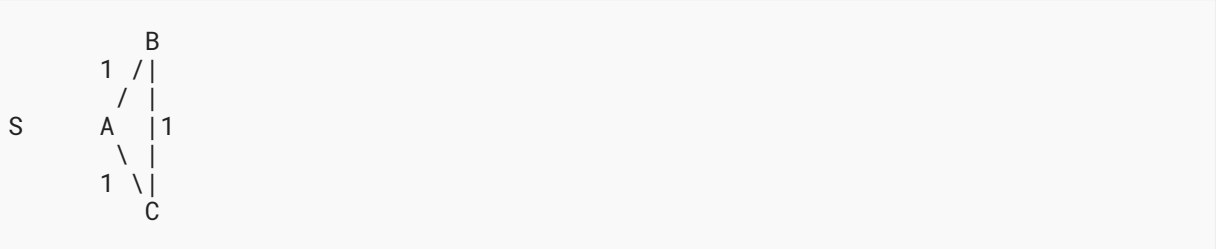
2.3. Transient Loops in Bellman-Ford

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.

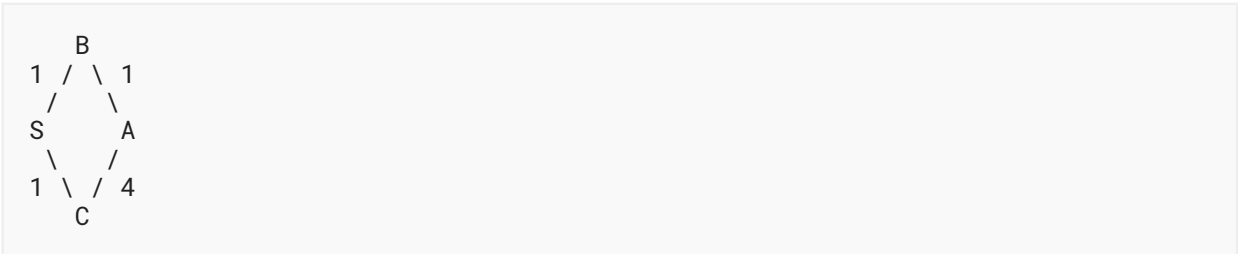
Many different feasibility conditions are possible. For example, BGP can be modelled as being a distance-vector protocol with a (rather drastic) feasibility condition: a routing update is only accepted when the receiving node's AS number is not included in the update's AS_PATH attribute (note that BGP's 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).

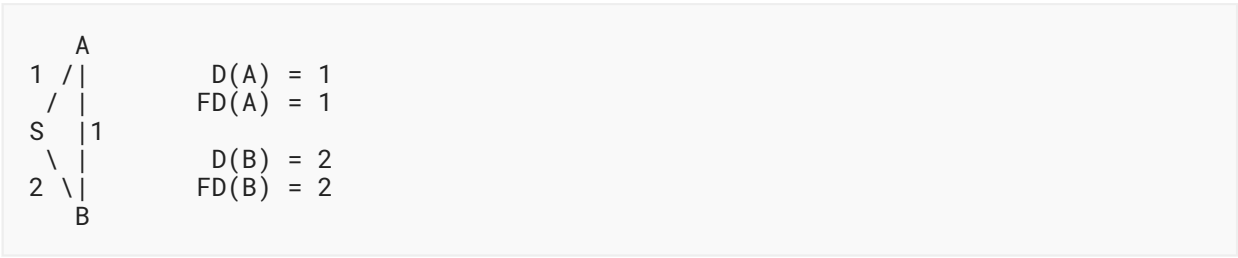
To see that it is strictly less restrictive, consider the following diagram, where A has selected the route through B, and D(A) = FD(A) = 2. Since D(C) = 1 < FD(A), the alternate route through C is feasible for A, although its metric C(A, C) + D(C) = 5 is larger than that of the currently selected route:



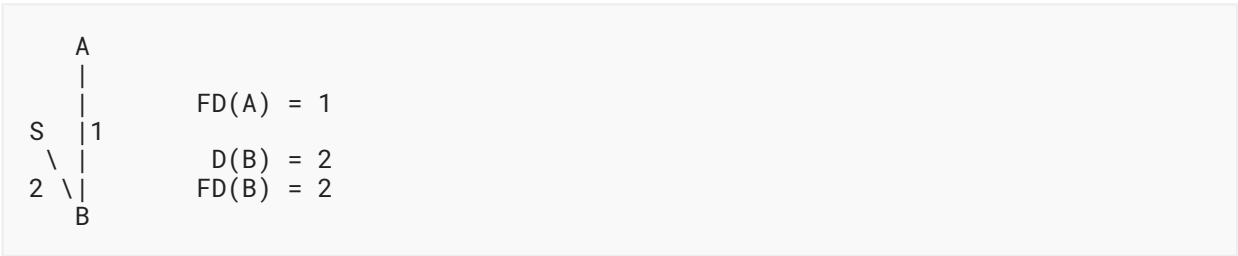
To show that this feasibility condition still guarantees loop-freedom, recall that at the time when A accepts an update from B, the metric D(B) announced by B is no smaller than FD(B); since it is smaller than FD(A), at that point in time FD(B) < FD(A). Since this property is preserved when A sends updates and also 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:

```
   A
   |
   |       FD(A) = (137, 1)
S  |1
 \ |        D(B) = (137, 2)
2 \|       FD(B) = (137, 2)
   B
```

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

```
   A
   |
   |       FD(A) = (137, 1)
S  |1
 \ |        D(B) = (138, 2)
2 \|       FD(B) = (138, 2)
   B
```

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 black-holes 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':

```
            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 3.5.4).

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). Each Babel packet consists of zero or more TLVs. Most TLVs may contain sub-TLVs.

Babel'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 3.8.1), 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 Acknowledgments (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 describe 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, 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 the following:

$$s + n \text{ (modulo } 2^{16}) = (s + n) \text{ MOD } 2^{16}$$

or, equivalently,

$$s + n \text{ (modulo } 2^{16}) = (s + n) \text{ AND } 65535$$

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 the following:

$$s < s' \text{ (modulo } 2^{16}) \text{ when } 0 < ((s' - s) \text{ MOD } 2^{16}) < 32768$$

or, equivalently,

$$s < s' \text{ (modulo } 2^{16}) \text{ when } s \neq s' \text{ and } ((s' - s) \text{ AND } 32768) = 0.$$

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 with each route: the route's distance, which is stored in the route table, and the feasibility distance, which is 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 to 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, at regular or irregular intervals, to indicate that it is alive. 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 nonzero 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 Expiry 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 for 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. 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); or
- 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 will 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 black-holes 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, the selection of a route increases congestion along it, which might increase packet loss, which in turn could cause its metric to increase; this kind of feedback loop 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: the strategy should only switch from the currently selected route 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 learnt 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 and 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 towards 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 the request if it is redundant (i.e., if the node 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.

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 10 ms, 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 (AEs):

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.

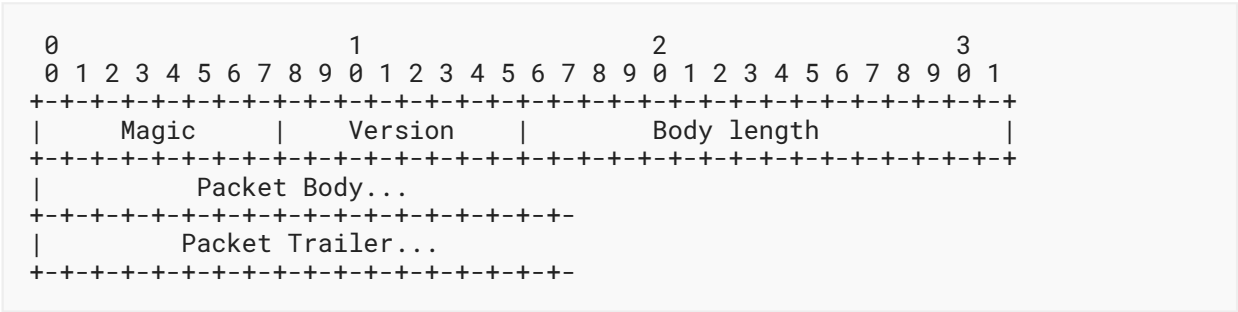
The address family associated with an address encoding is either IPv4 or IPv6: it is undefined for AE 0, IPv4 for AE 1, and IPv6 for AEs 2 and 3.

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).

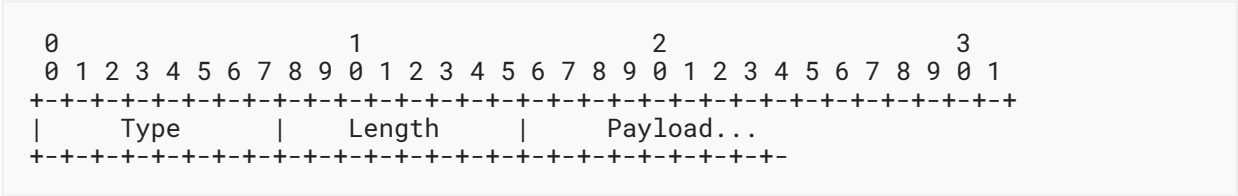
Packet Body The packet body; a sequence of TLVs.

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.

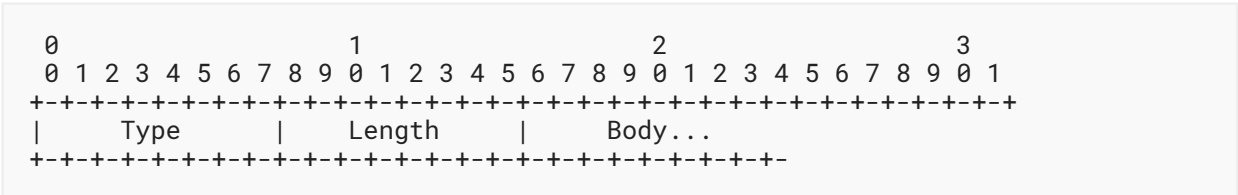
Payload The TLV payload, which consists of a body and, for selected TLV types, an optional list of sub-TLVs.

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 the 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, the Router-Id TLV itself may be omitted, and the router-id is 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:

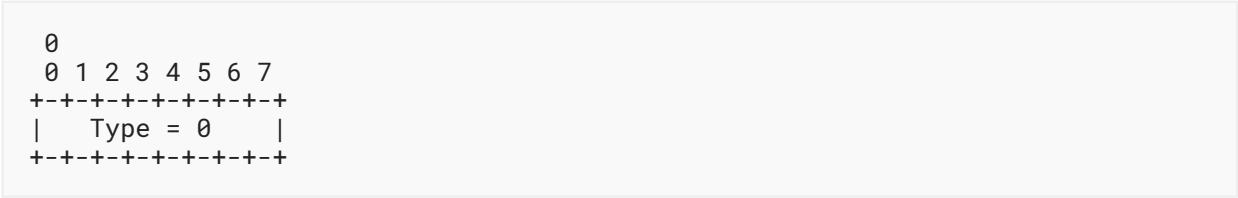
- for each address encoding that allows compression, the current default prefix: this is undefined at the start of the packet and is updated by each Update TLV with the Prefix flag set (Section 4.6.9);
- for each address family (IPv4 or IPv6), the current next hop: this is the source address of the enclosing packet for the matching address family at the start of a packet, and it 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 it 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

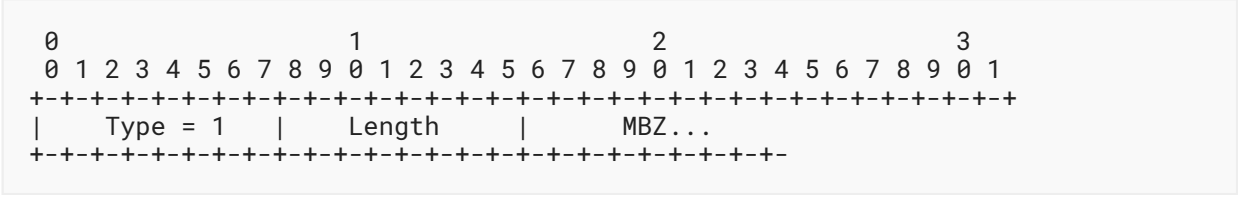


Fields:

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:

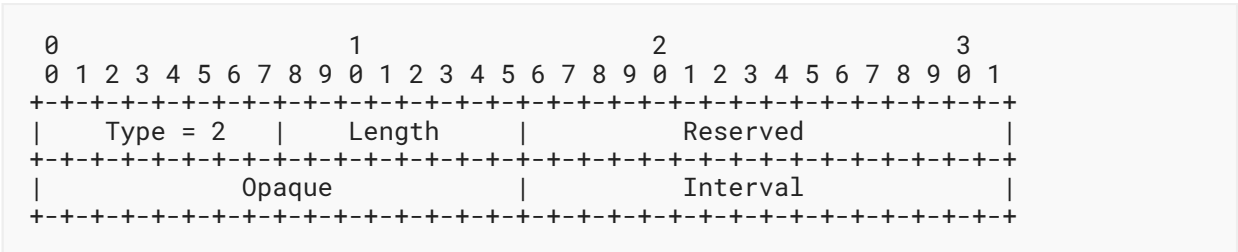
Type Set to 1 to indicate a PadN TLV.

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.

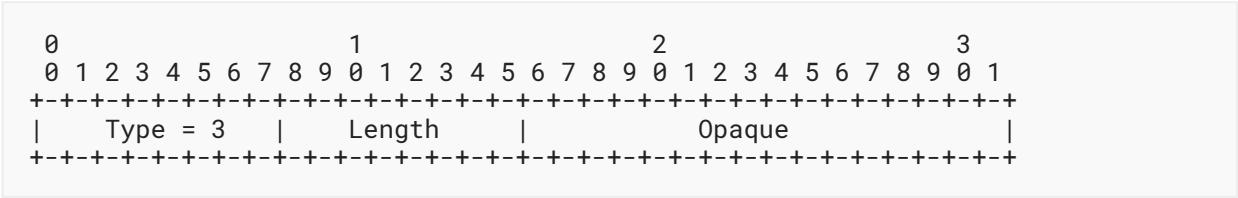
Reserved Sent as 0 and **MUST** be ignored on reception.

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 TLV.

Fields:

Type Set to 3 to indicate an Acknowledgment TLV.

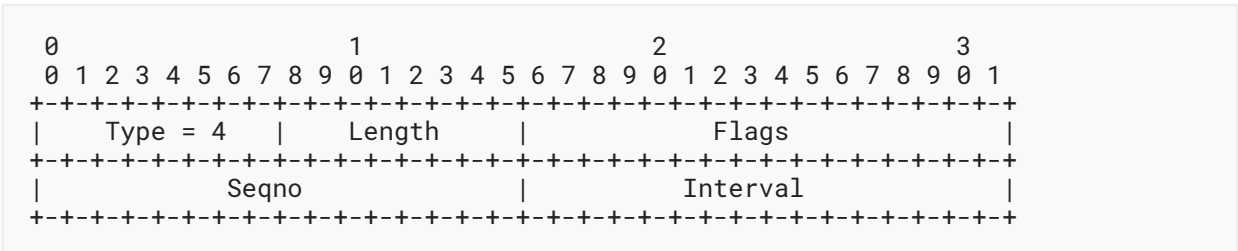
Length The length of the body in octets, exclusive of the Type and Length fields.

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.

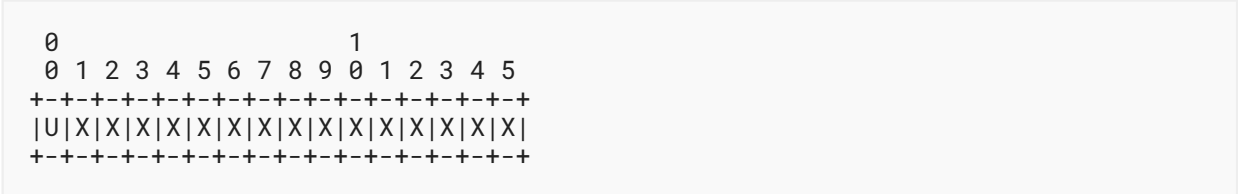
Length The length of the body in octets, exclusive of the Type and Length fields.

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 nonzero, 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:



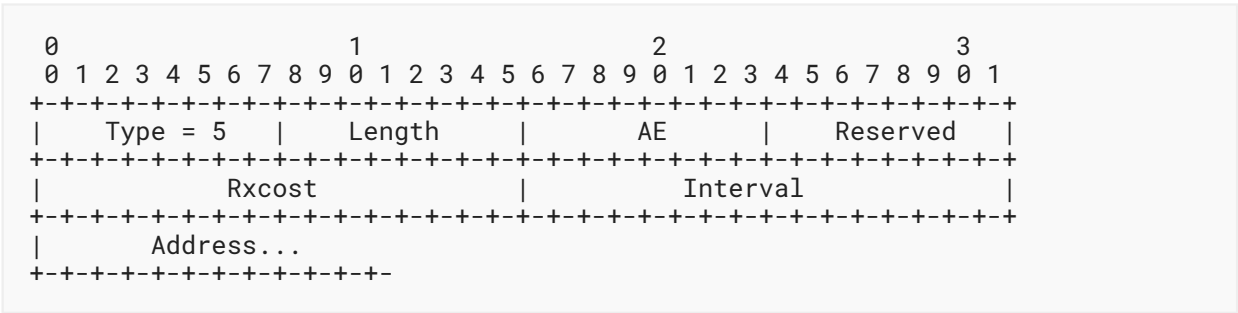
U (Unicast) flag (8000 hexadecimal): if set, then this Hello represents a Unicast Hello, otherwise it represents a Multicast Hello;

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 neighbours 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.

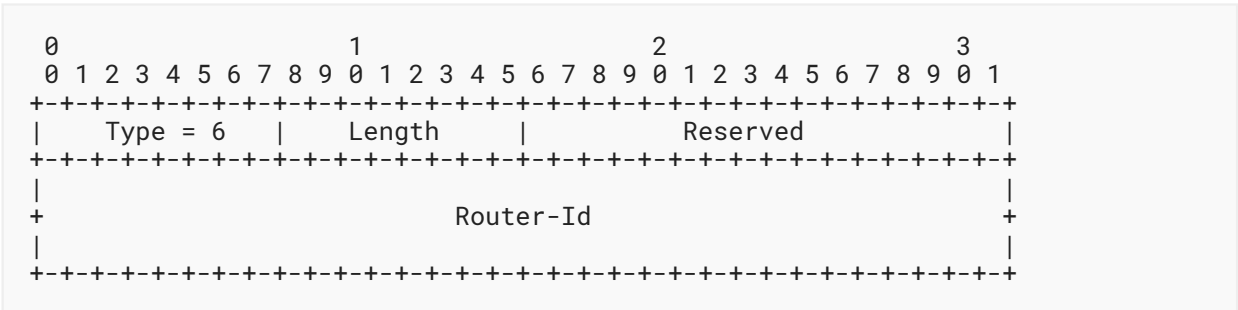
Address The address of the destination node, in the format specified by the AE field. Address compression is not allowed.

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.

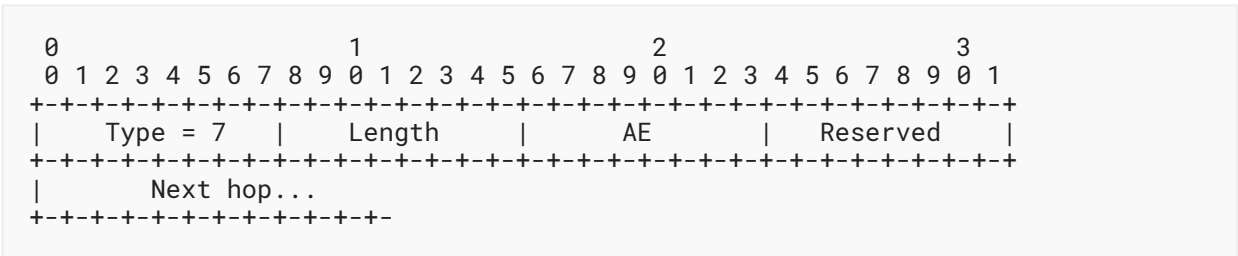
Length The length of the body in octets, exclusive of the Type and Length fields.

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.

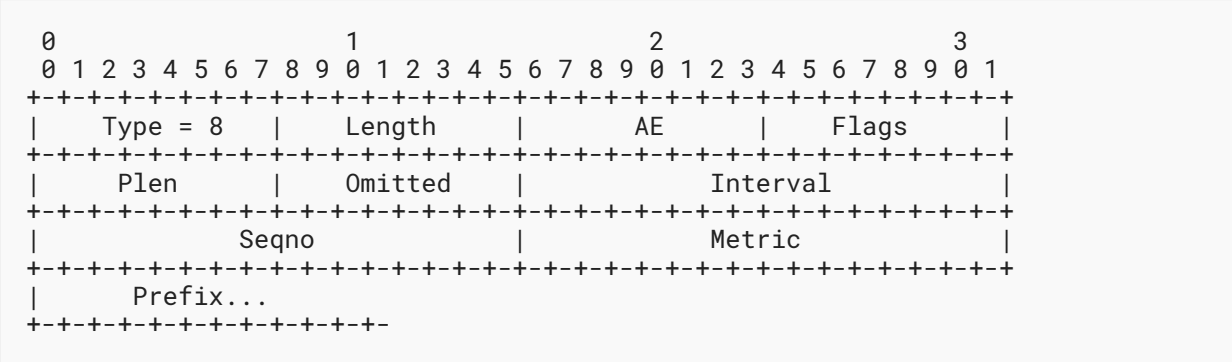
Next hop The next-hop address advertised by subsequent Update TLVs for this address family.

When the address family matches the network-layer protocol over which this packet is transported, 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), the Omitted field **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:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|P|R|X|X|X|X|X|X|
+-+-+-+-+-+-+-+-+
```

P (Prefix) flag (80 hexadecimal): if set, then this Update TLV 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 the Omitted field 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.

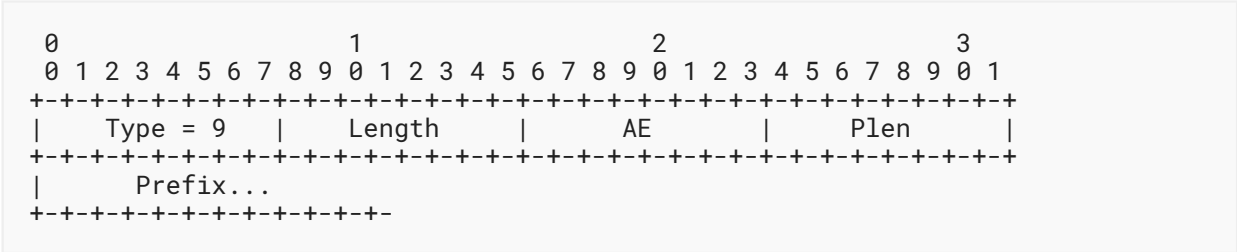
The next-hop address for this update is taken from the last preceding Next Hop TLV with a matching address family (IPv4 or IPv6) in the same packet even if it was otherwise ignored due to an unknown mandatory sub-TLV; if no such TLV exists, it is taken from the network-layer source address of this packet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

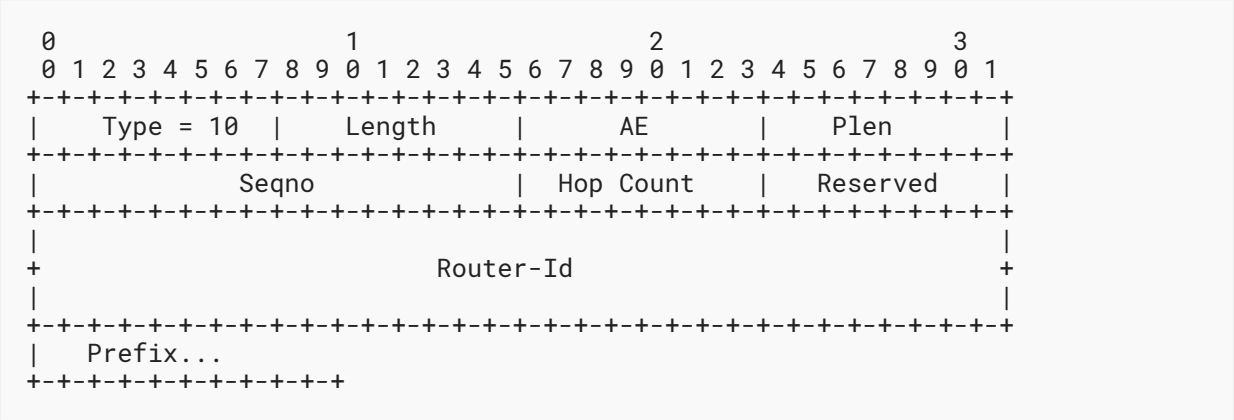
Plen The length in bits of the requested prefix.

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.

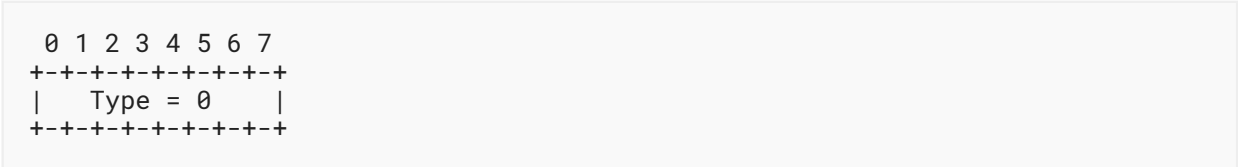
A Seqno Request TLV prompts the receiving node to send a finite-metric Update for the prefix specified by the AE, Plen, and Prefix fields, with either a router-id different from what is specified by the Router-Id field, or a Seqno no less (modulo 2^{16}) than what is specified by the Seqno field. If 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

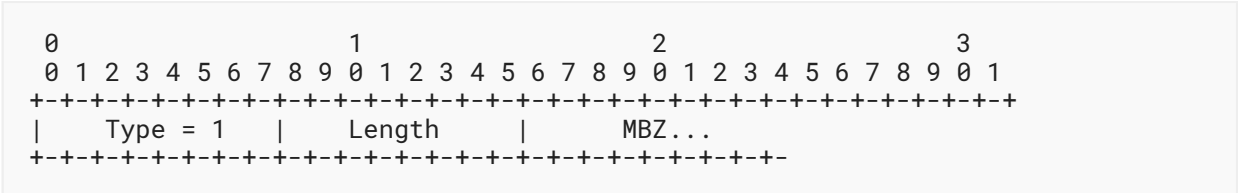


Fields:

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	RFC 8966

Type	Name	Reference
1	PadN	RFC 8966
2	Acknowledgment Request	RFC 8966
3	Acknowledgment	RFC 8966
4	Hello	RFC 8966
5	IHU	RFC 8966
6	Router-Id	RFC 8966
7	Next Hop	RFC 8966
8	Update	RFC 8966
9	Route Request	RFC 8966
10	Seqno Request	RFC 8966
11	TS/PC	[RFC7298]
12	HMAC	[RFC7298]
13	Reserved	
14	Reserved	
15	Reserved	
224-254	Reserved for Experimental Use	RFC 8966
255	Reserved for expansion of the type space	RFC 8966

Table 1

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	RFC 8966
1	PadN	RFC 8966
2	Diversity	[BABEL-DIVERSITY]

Type	Name	Reference
3	Timestamp	[BABEL-RTT]
4-111	Unassigned	
112-126	Reserved for Experimental Use	RFC 8966
127	Reserved for expansion of the type space	RFC 8966
128	Source Prefix	[BABEL-SS]
129-239	Unassigned	
240-254	Reserved for Experimental Use	RFC 8966
255	Reserved for expansion of the type space	RFC 8966

Table 2

IANA has created 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	RFC 8966
1	IPv4 address	RFC 8966
2	IPv6 address	RFC 8966
3	Link-local IPv6 address	RFC 8966
4-223	Unassigned	
224-254	Reserved for Experimental Use	RFC 8966
255	Reserved for expansion of the AE space	RFC 8966

Table 3

IANA has renamed the registry called "Babel Flags Values" to "Babel Update Flags Values". The allocation policy for this registry is Specification Required. The values in this registry are as follows:

Bit	Name	Reference
0	Default prefix	RFC 8966

Bit	Name	Reference
1	Default router-id	RFC 8966
2-7	Unassigned	

Table 4

IANA has created 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	RFC 8966
1-15	Unassigned	

Table 5

IANA has replaced all references to RFCs 6126 and 7557 in all of the registries mentioned above with 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, black-holed, 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 trade-offs between implementation simplicity and security:

- Babel over DTLS [RFC8968] runs the majority of Babel traffic over DTLS and leverages DTLS to authenticate nodes and provide confidentiality and integrity protection;
- MAC authentication [RFC8967] 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 [RFC8967]). 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. References

7.1. 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>.

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, 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, 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>.

[RFC8967] Dô, C., Kolodziejak, W., and J. Chroboczek, "MAC Authentication for the Babel Routing Protocol", RFC 8967, DOI 10.17487/RFC8967, January 2021, <https://www.rfc-editor.org/info/rfc8967>.

7.2. Informative References

[BABEL-DIVERSITY] Chroboczek, J., "Diversity Routing for the Babel Routing Protocol", Work in Progress, Internet-Draft, draft-chroboczek-babel-diversity-routing-01, 15 February 2016, <https://tools.ietf.org/html/draft-chroboczek-babel-diversity-routing-01>.

[BABEL-RTT] Jonglez, B. and J. Chroboczek, "Delay-based Metric Extension for the Babel Routing Protocol", Work in Progress, Internet-Draft, draft-ietf-babel-rtt-extension-00, 26 April 2019, <https://tools.ietf.org/html/draft-ietf-babel-rtt-extension-00>.

[BABEL-SS] Boutier, M. and J. Chroboczek, "Source-Specific Routing in Babel", Work in Progress, Internet-Draft, draft-ietf-babel-source-specific-07, 28 October 2020, <https://tools.ietf.org/html/draft-ietf-babel-source-specific-07>.

[DSDV] Perkins, C. and P. Bhagwat, "Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers", ACM SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications, 234-244, DOI 10.1145/190314.190336, October 1994, <https://doi.org/10.1145/190314.190336>.

[DUAL] Garcia Luna Aceves, J. J., "Loop-free routing using diffusing computations", IEEE/ACM Transactions on Networking, 1:1, DOI 10.1109/90.222913, February 1993, <https://doi.org/10.1109/90.222913>.

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

[ETX] De Couto, D., Aguayo, D., Bicket, J., and R. Morris, "A high-throughput path metric for multi-hop wireless networks", MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking, 134-146, DOI 10.1145/938985.939000, September 2003, <https://doi.org/10.1145/938985.939000>.

[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, <https://doi.org/10.1109/ieeestd.2012.6178212>.

[IEN137] Cohen, D., "On Holy Wars and a Plea for Peace", IEN 137, 1 April 1980.

[IS-IS] International Organization for Standardization, "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, DOI 10.1109/90.298431, April 1994, <https://doi.org/10.1109/90.298431>.

[OSPF] Moy, J., "OSPF Version 2", STD 54, RFC 2328, DOI 10.17487/RFC2328, April 1998, <https://www.rfc-editor.org/info/rfc2328>.

[PACKETBB] Clausen, T., Dearlove, C., Dean, J., and C. Adjih, "Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format", RFC 5444, DOI 10.17487/RFC5444, February 2009, <https://www.rfc-editor.org/info/rfc5444>.

[RFC2675] Borman, D., Deering, S., and R. Hinden, "IPv6 Jumbograms", RFC 2675, DOI 10.17487/RFC2675, August 1999, <https://www.rfc-editor.org/info/rfc2675>.

[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, <https://www.rfc-editor.org/info/rfc6126>.

[RFC7298] Ovsienko, D., "Babel Hashed Message Authentication Code (HMAC) Cryptographic Authentication", RFC 7298, DOI 10.17487/RFC7298, July 2014, <https://www.rfc-editor.org/info/rfc7298>.

[RFC7557] Chroboczek, J., "Extension Mechanism for the Babel Routing Protocol", RFC 7557, DOI 10.17487/RFC7557, May 2015, <https://www.rfc-editor.org/info/rfc7557>.

[RFC8968] Décimo, A., Schinazi, D., and J. Chroboczek, "Babel Routing Protocol over Datagram Transport Layer Security", RFC 8968, DOI 10.17487/RFC8968, January 2021, <https://www.rfc-editor.org/info/rfc8968>.

[RIP] Malkin, G., "RIP Version 2", STD 56, RFC 2453, DOI 10.17487/RFC2453, November 1998, <https://www.rfc-editor.org/info/rfc2453>.

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 Expected Transmission Cost (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 our 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 with 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 Hello history 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 estimates 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 10 ms, 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 nonmandatory 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 nonmandatory sub-TLV, the TLV is parsed and acted upon, and only the unknown sub-TLV is silently ignored. Therefore, a nonmandatory 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 dual-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 20,000-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 re-announce any routes that it has learnt from its neighbours, and always prefers the direct route to a directly attached prefix to a route learnt 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 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 1,000 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 misinterpret a Unicast Hello for a Multicast one; since the sequence number space of Unicast Hellos is distinct from the sequence number 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 either to ignore or to 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).

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 Høiland-Jørgensen, Benjamin Kaduk, Joao Sobrinho, and Martin Vigoureux. The previous version of this specification [RFC6126] greatly benefited from the input of Joel Halpern. The address compression technique was inspired by [PACKETBB].

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
United States of America
Email: dschinazi.ietf@gmail.com