2. Background

This section summarizes the aspects of Babel and ns-3 most relevant to this paper.

2.1. Babel

Babel was first published as an Experimental RFC in 2011 [7] and later in January 2021 as a Standards Track RFC [1]. Babel is a robust, proactive, loop-avoiding distance-vector routing protocol, and is suitable for both wired and wireless networks. Being a proactive routing protocol implies it preemptively exchanges routing information for all prefixes, whether there are any packets to be routed or not. A routing loop is a phenomenon where packets get routed in a circle, thereby using up bandwidth while not reaching their destination. In many routing protocols routing loops can form. However, the Babel specification guarantees that no routing loops will ever form if each prefix is originated by only one router. In the case that some prefixes are originated by multiple routers, the specification guarantees that all routing loops quickly disappear and the same loop can never form again [1].

Protocol Operation. In the Babel protocol, routers exchange packets as UDP datagrams sent to a specific neighbor or the multicast address specified in the RFC with a hop count of one. Each packet may contain several messages, called TLVs (Type-Length-Value). There are 11 different types of TLVs in the RFC. However, the protocol is extensible and allows for more types of TLVs to be added. Additionally, most TLVs can contain sub-TLVs.

For neighbor discovery and link quality estimation, each node periodically sends Hello TLVs. Each node computes the receiving cost for each neighbor based on the number of received and missed Hellos from that neighbor. There are two cost computation strategies suggested in the RFC. For wired links, the receiving cost is either a constant value (chosen by the implementation) if the last k out of j Hellos were received, or infinity otherwise. In contrast to wired links, wireless links are not just up or down but have a larger range of link qualities. Therefore the expected transmission count (ETX) metric is suggested: the receiving cost is a constant divided by the fraction of recently correctly received Hellos. This metric has the advantage to favor short, stable links over long, lossy links, which a hop-count-based metric favors [1], [8]. The node regularly sends this receiving cost back to
the neighbor inside an IHU TLV (I Heard You). This is necessary since links are not generally symmetric.

Babel can carry prefixes and is, therefore, able to do prefix-based routing. However, its design assumes that each node has a full routing table (to all of the nodes in the network) and is therefore well suited for mesh routing.

The protocol has specific optimizations used on symmetric wired links. For example, it will not resend an update received on a point-to-point link on the same link.

To ensure the strict properties regarding routing loops described above, Babel combines concepts from different routing protocols. The idea Babel uses to (almost) entirely avoid routing loops is the concept of feasibility. Each node maintains for each source (prefix and its originating router) a feasibility distance. This "distance" consists of the newest seqno (sequence number) of the route and the best distance the node has ever announced for this source with the current seqno.

When receiving an update from an Update TLV, a router checks whether the received route has either a newer sequence number or a smaller metric than any it has ever announced. If so, it can be sure not to cause a routing loop by switching to this route. When the topology changes, it might be the case that a router has no feasible routes left. In that case, it sends a seqno request, triggering the source of the route to increment its seqno. After incrementing the seqno, the new route gets forwarded to the router that sent the seqno request [1].

Applications. Babel routers exchange routing information even when there is no mobility event, thus potentially generating unnecessary traffic. Therefore Babel is not the ideal choice for routing in some situations such as large and stable networks and low-power networks. However, Babel is a robust protocol and can be successfully used in most environments. The most prominent use cases include small home networks, heterogeneous networks, and mesh networks [8].

Performance. For the application of mesh networks, multiple experiments conclude that Babel's performance is at least comparable - if not better - than other mesh routing protocols such as OLSR and BATMAN [9]–[12].

2.2. ns-3

ns-3 is an open-source network simulator first published in 2008 as the successor to the popular network simulator ns2 [13]. It is one of the most widely used network simulators serving as a tool to many network researchers. Like most network simulators, ns-3 is a discrete, event-based simulator; the simulation time is stepped from one event to the next and at each step, all necessary calculations are performed. In terms of both memory usage and computation time, ns-3 is a highly performant simulator capable of large-scale simulations with hundreds or thousands of nodes. It tries to provide a realistic simulation of all network components such as the IP stack or network devices [14].

ns-3 is written in C++ and is structured into modules responsible for different aspects of the simulation. Each with its own tests, examples, and documentation.

3. Implementation

This section outlines the most important aspects of our implementation of the Babel routing protocol for ns-3. Since it is recommended in the Babel RFC to route all control traffic via IPv6, the protocol is implemented as an IPv6 routing protocol.

The implementation is written in C++ and the structure is partially based on the OLSR module included in ns-3 [6]. We provide a new module called babel consisting of a simple example network, a helper class to install Babel on existing ns-3 nodes, and the implementation of the protocol itself. The main functionality is located inside the ns3::babel::RoutingProtocol class, which extends ns3::Ipv6RoutingProtocol. To route IPv6 packets, the methods RouteOutput and RouteInput are called by the ns-3 IP-stack for outbound and inbound packets respectively. The ns3::babel::PacketHeader and ns3::babel::TLV classes are responsible for serializing and deserializing Babel packets and the TLVs contained inside them.

ns-3 includes a TypeId feature used by the helper class to construct protocol instances. We can add attributes with default values to our TypeId, which are used to initialize the objects. This feature is essential since it allows the creator of a simulation to set specific protocol parameters for all nodes or individual nodes. The default values are taken from the RFC. Tunable protocol parameters are, for example, the time intervals used for sending scheduled Hello, IHU, and Update TLVs, as well as the (urgent) timeout for sending messages. The periodic sending of Hello, IHU, and Update TLVs is governed by Timers set to times specified by the attributes. When a timer expires, we queue the required TLVs for sending on each interface.

To aggregate multiple TLVs into one packet and to apply randomization to the timing of messages, we keep track of a list of queued TLVs and a timer for each interface. Instead of sending a message directly, we instead add it to the queue for later sending and set the timer if it was not already (the timer duration is random within a range specified by the attributes). When the timer expires, it calls a method for sending the packet. This mechanism also allows for the sending of "urgent TLVs" within a shorter timeout. Upon queuing an urgent TLV, the timer is rescheduled to be inside the "urgent timeout" (if it was not already).

The protocol encoding optimizes the size of the packets by not sending redundant information inside each TLV. For example, the Update TLV might not contain the router-id of the router originating this particular route update but relies on a Router-Id TLV preceding it. To follow the encoding, we need to keep track of a parser state for incoming and outgoing packets. Therefore, alongside the queue and timer, we keep track of the parser state of the outgoing packet for each interface. This allows, for example, an Update TLV to add a Router-Id TLV if there was no Router-Id TLV yet or the last Router-Id TLV contained a different router-id.

To receive packets, there is a receiving UDP socket set to listen on the specified multicast address. When a UDP datagram arrives (either as multicast or unicast), the Babel packet in the datagram is delivered
by the ns-3 IP-stack to a callback method inside the
ns3::babel::RoutingProtocol class. Inside this call-
back method, we deserialize the packet and, while keeping
track of the parser state, loop over all TLVs contained
inside. If there are any TLVs that require the selected
routes to be recomputed, this is done once after all TLVs
are processed. The recomputation of the routes may lead
to the queuing of new Update TLVs.

The nodes compute their receiving cost using the
ETX algorithm outlined in Section 2.1 as the strategy for
wireless links.

When routing packets, we need to find the route for
the longest matching prefix of the destination address. The
route table is a map with the prefix as the key. To allow
for fast lookups of routes based on their prefix. Since we
do not know the length of the longest prefix and looping
over all 128 possible could be costly, we maintain a list
of all of the prefix lengths we currently store and loop over
it instead. A further optimization would be to maintain a
tree structure for finding the longest matching prefix or
cache route lookups.

Furthermore, we may know of multiple routes for one
prefix and, although only one is selected, it is necessary to
keep track of the other ones as fallback routes. Maintain-
ing (for each prefix) a list of routes with a pointer to the
currently selected one solves this problem while keeping
fast access to the selected route.

During the simulation setup in ns-3, when installing
the Babel routing protocol on a node using the Babel
helper class, it is possible to exclude interfaces from the
Babel protocol. This way, a network of Babel nodes can
link to the other nodes, possibly using another routing
protocol. All Babel nodes originate all of their global
addresses as well as the prefixes of the excluded interfaces.
To illustrate, consider the network in Figure 1, where R
and A are Babel nodes and S is another server.

![Figure 1: Network Topology](image)

During the setup of R, its left interface (to S) has been
excluded from Babel routing. Therefore, R originates the
prefix of this network (2001:1::/64) along with its global
addresses.

### 3.1. Capabilities

Our implementation is a working IPv6 routing proto-
col for ns-3. Considering the only other routing protocol
for IPv6 is currently ns3::Ipv6StaticRouting, just hav-
ing a non-static routing protocol might already be helpful

The main point of the implementation, and hence its
most relevant capability, is to accurately depict the behav-
or of Babel inside ns-3. This is achieved by following the
specification and further demonstrated in Section 4.

The use of ns-3 attributes makes it easy to tune proto-
col parameters to meet the needs of a specific simulation.
A performance comparison with different protocol param-
eters is also possible.

Since ns-3 has the option to trace all packets to a
file and we serialize all packets as described by the
specification, it is possible to use a tool such as Wireshark
to inspect Babel packets exchanged during a simulation.

### 3.2. Limitations

The goal of the current implementation is to provide a
working version of Babel for simulations of the protocol
behavior in different environments. As of now, it is lacking
some features and does not yet comply with all aspects of
the RFC. Most aspects of non-compliance are not an issue
since they are not strictly required for the protocol, and
our protocol instances only communicate with other nodes
inside ns-3 using the identical implementation. In other
words, for simulations inside ns-3, no interoperability with
other protocol implementations is needed.

Most notably, the routing protocol is currently an
IPv6 routing protocol and only supports IPv6 traffic.
The recommended way to use Babel is to have a single
protocol instance that routes IPv4 and IPv6 traffic but
communicates exclusively using IPv6 [1]. Such behavior
can most likely be achieved in ns-3. However, since most
simulations use either IPv4 or IPv6 it would be desirable
to have a standalone IPv4 implementation as well.

There is currently no recognition of Sub-TLVs, Uni-
cast Hellos, Acknowledgments, Acknowledgment Re-
quests and some encoding methods as defined in the RFC.
As mentioned before, if this implementation does not need
to interoperate with others this is not a problem.

### 4. Tests

To demonstrate the functionality of the implementa-
tion, we devised a test scenario. The setup consists of 6
nodes connected on six point-to-point links as shown in
Figure 2. Nodes R, A, B, C, and D are Babel routers. After
allowing the protocol a brief initialization time, nodes A,
B, C, and D start emitting 20 UDP packets per second to
S. At 35 seconds into the simulation they stop sending
the packets. At 20 seconds, the link between R and A gets
cut. To route packets to S, the routers use their routes to
2001:6::/64, a prefix originated by R.

![Figure 2: Network topology under test](image)

Node S tracks the number of packets arriving, and its
results are shown in Figure 3. While this graph illustrates
the routes leading to S, the protocol also tracks all other
routes, which are not visualized here. For clarity, the
packets from C and D are not shown since they are not
affected by the link being cut and all packets arrive as
expected.
Figure 3: Packets Received at S

Just before cutting the link, node B has two routes to 2001:6::/64 (and therefore S). One route via C and another via A. When A detects the link is cut, it sends a route retraction for all routes it would forward to R. When receiving this retraction, B switches to its route via C and sends an update. A can, however, not switch to this route via C, as it is not feasible. Essentially, A has no way of knowing that adopting this route would not create a routing loop. Left with no routes, A sends a seqno request, which gets forwarded by B, C, and D, until it reaches R. R increments its seqno and sends an update with the new seqno, which is quickly forwarded back to A. After receiving an update with the new seqno, A can now switch to a route via B. During the time it takes the seqno request and updates to go around the network, A has no route to 2001:6::/64 (and therefore S). This can be seen in Figure 3 as the drop in packets received from A. The increased number of packets traveling to S via D is still below the capacity of the links, so there is no drop in the packets received from the other nodes.

And the end, the route table of A for routes to 2001:6::/64 looks as follows (advertised metric is the metric announced by the neighbor). The cost of a route is the advertised metric plus the cost of the link to the neighbor:

- next hop: fe80::200:ff:fe00:4; advertised metric: 768; seqno: 0x8000
- next hop: fe80::200:ff:fe00:1; advertised metric: 0; seqno: 0x8000

fe80::200:ff:fe00:4 is the link-local address of an interface of B. This is the selected route with a seqno one higher than the other route. fe80::200:ff:fe00:1 is the link-local address of an interface of R. Although the advertised metric is 0, the metric overall is infinity since the link cost from A to R is infinity.

5. Conclusion and Future Work

This paper introduced an implementation of the Babel routing protocol for the discrete event simulator ns-3. This implementation can be used to help research applications and the performance of Babel using ns-3. Although the current state of our work suffices to simulate the operation of Babel, it is desirable to finish the implementation to comply with the RFC (see Section 3.2).

To fulfill the goal of providing an easy way to simulate the behavior of the Babel protocol, the behavior inside the simulation must match the behavior in the real world. Therefore, it is vital to validate the results from the simulation with results obtained in the real world. This requires either carrying out hardware tests or recreating an existing test setup inside the simulator.

An interesting idea, which would be easy to test now, is writing an extension to optimize protocol performance in fast-moving mobile ad-hoc networks by relaying position information through the protocol. A similar idea using a custom OLSR implementation shows promising results in [16], and it is interesting to see how that compares to a Babel version.

References