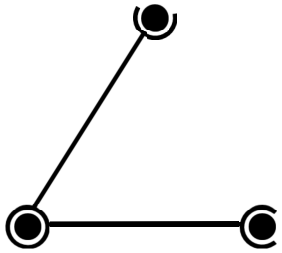

Babel

A delay-based metric for the Babel routing protocol

Internship at PPS, Université Paris Diderot

Author:
Baptiste Jonglez
(ENS Lyon)

Supervisors:
Juliusz Chroboczek
Matthieu Boutier

L3 Informatique Fondamentale
(3rd year licence in CS)

August 25, 2013

Contents

Introduction

This document is a report of my internship at PPS (Proofs, Programs and Systems), a joint lab of the CNRS (Centre National de la Recherche Scientifique) and the Université Paris Diderot. The internship lasted 7 weeks, from 3rd June to 19th July 2013. It was supervised by Juliusz Chroboczek, Maître de Conférences at the Université Paris Diderot, and co-supervised by Matthieu Boutier, who is currently doing a Ph.D under Juliusz's supervision.

The goal of my internship was to adapt Babel, a routing protocol developed by Juliusz, to a new kind of network: *overlay networks*. This work was motivated by the needs of Nexedi, a company using Babel for its cloud infrastructure.

The key parameter in an overlay network is the *delay* between routers. In order to use the delay with Babel, I worked on an algorithm proposed by Juliusz, which had been initially developed in the 80s but long forgotten. This led me to implement the algorithm in `babeld`, the reference implementation of Babel. This work also required a new extension to the Babel protocol, which I co-designed and documented.

The last part of the internship was focused on evaluating my implementation. Thanks to Nexedi, I was able to experiment on a real (albeit small) overlay network, with some routers located as far as Tokyo. I also setup simulated networks, which makes it easier to stress-test the algorithm in complex or unusual situations. This evaluation suggests that my implementation is production-ready.

I would like to acknowledge Juliusz and Matthieu for their excellent supervision, and Gabriel Kerneis for his occasional help. Various people on the `babel-users` mailing list provided valuable comments on this work, which was highly appreciable. Lastly, thanks to the other Ph.D students at PPS, for the day-to-day life at the lab (and for the excellent tea left as a self-service).

1 Background

In this section, we provide some background on overlay networks, which were the motivation behind our work, and routing in general. The last subsection covers delay and RTT in a network, which is the basis for the algorithm described in Section 2.4.

1.1 Overlay networks

With the advent of affordable, reliable and fast Internet access, building virtual networks on top of the Internet is becoming increasingly easy.

One of the incentive for our work was Nexedi's needs. Nexedi[1] is a company specialised in cloud management. It has given birth to SlapOS[2], a decentralised cloud operating system that can run on servers located anywhere (datacenters around the world, house, etc). To interconnect all nodes of a SlapOS cloud, IPv6 is used; however, most ISPs provide unreliable IPv6 connectivity, which hampers the availability of the cloud[3].

To solve this reliability issue, Nexedi has made use of *overlay networks*. The nodes connect to each other through *tunnels*, which are point-to-point virtual links over the IPv4 Internet. IPv6 is then tunnelled through these virtual links, so that it doesn't use the unreliable IPv6 connectivity provided by commercial ISPs. To keep a scalable system, the network is not fully meshed[4], and the Babel routing protocol (see Section 1.4) is used inside the resulting overlay network. The software written to create and manage such an overlay network is called `re6st`. [1]

1.2 Routing in a network

In order to reach other nodes, a node must be able to find *routes* through the network.

The classical routing paradigm is *hop-by-hop routing*. In this model, each router maintains a *routing table*. This table contains (`destination`, `next-hop`) pairs, where `destination` is a network prefix, and `next-hop` is the address of a neighbouring router. Each entry indicates that, to reach nodes in `destination`, the router must forward the data to `next-hop`.

To achieve proper routing, the routing table of each router must be populated and maintained. This can be achieved either manually, or automatically, by using a routing daemon.

---

[1] `http://www.nexedi.com`

[2] `https://www.slapos.org/`

[3] `http://lists.alioth.debian.org/pipermail/babel-users/2013-January/001132.html`

[4] In a fully meshed network, all nodes have direct links to each other, which forms a complete graph.

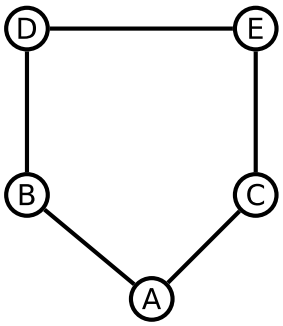

Figure 1: Simple network with 5 nodes.

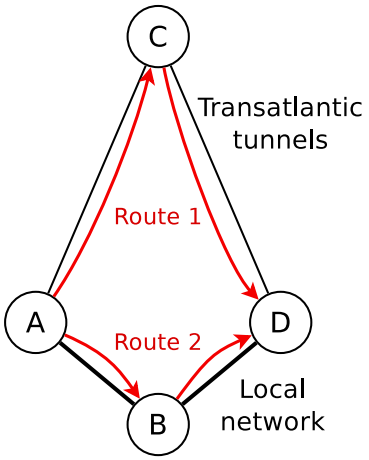

Figure 2: Route 2 should be preferred.

Static routing The most elementary way to achieve routing is by manually configuring each node's routing table. With a clear vision of the network topology, a human operator can decide which route to use for each source and destination. For instance, on the network of Figure 1, the operator could decide "From A, use B as a next-hop to reach D or B. To reach C or E, use C as a next-hop", and similar rules for each source node.

However, this method is tedious and lacks flexibility, even for small networks. Each time a link goes down, the operator has to manually update the routing tables to route around the faulty link. Each time a node is added to the network, all other nodes' routing tables must be updated to know how to reach the new node.

Routing algorithm A *routing algorithm* can do the job of the human operator. By having all the nodes run the routing algorithm, they can automatically discover the network layout. The nodes are then able to automatically build their routing table, based on information provided by other nodes thanks to the routing algorithm.

Thus, using a routing algorithm saves the burden of manual configuration. Additionally, the routing tables react dynamically to changes in the network, for instance when links are added or removed.

Many classical routing protocols have been developed over the past decades [10, 13, 5] . In the recent years, new routing protocols have been designed for ad-hoc wireless mesh networks [3] , because this environment is much more dynamic and brings some additional difficulties. Babel, the routing protocol on which this work is based, belongs to this newer class of protocols.

1.3 The importance of metrics

Routing algorithms are often quite generic, since a given routing algorithm can be used on a wide range of networks. As an analogy, consider a GPS device. It

computes shortest paths, but it must be told on which ground. The user may chose between the fastest path, the least expensive one, or a path that avoids highways.

For routing protocols, a crucial parameter is the way of doing *link quality estimation*. Each kind of network has specific characteristics, and the notion of "good" and "bad" links must be defined accordingly. For instance, the *capacity* of the links can be a criteria: for a typical ISP, a 10G link is better than a 100M link. Another criteria could be to avoid inter-continental links.

With the proper policy for defining the *quality* of the links, the same routing protocol can be used on very different networks.

Cost and metric Whatever policy is considered, it has to be understood by the routing protocol. The policy is typically turned into a *cost* associated with a given link. The higher the cost, the less attractive the link appears to the routing protocol.

Given a route between any two nodes, the *metric* of the route is the sum of the costs of all the links along the route.[5] The cost of a link can be assigned manually, or can be dynamically derived from other parameters: link usage, packet loss, and so on.

Note that costs and metrics need not be integers. Other algebras can be used; a description of a general algebra suitable for routing problems can be found in [16].

Hopcount metric The simplest metric is the "hopcount" metric, where each link has a unit cost. Therefore, a node will try to minimise the number of *hops* (routers crossed on the path) to reach a destination.

While this metric is satisfactory for a purely wired network, it falls short in more complicated network, where links are not all equivalent. A limitation of the hopcount metric in an overlay network is shown on Figure 2. The hopcount metric is as likely to use the $A \to B \to D$ route (route 1) as the $A \to C \to D$ route (route 2), while we definitely want to avoid the transatlantic tunnels.

On wireless mesh networks, it has been shown that the hopcount metric is the worst possible. Indeed, it tends to choose the longest links in term of physical distance, which are also the worst in term of stability and capacity.

1.4 The Babel routing protocol

Babel is a loop-avoiding distance-vector routing protocol for IPv6 and IPv4 with fast convergence properties. It is based on the ideas in DSDV, AODV and Cisco's EIGRP, but is designed to work well not only in wired networks but also in wireless mesh networks.[6]

[5] This paragraph is borrowed from [3].

[6] Excerpt from `http://www.pps.univ-paris-diderot.fr/~jch/software/babel/`, retrieved 19th August 2013

While optimised for wireless networks, Babel also works on other types of links. It is thus possible to use Babel in a overlay network.

The Babel protocol is easily extensible [4], which makes it a good choice for experimenting with a new kind of metric.

1.5 Delay and RTT

Given two hosts A and B on a network, the *one-way delay* from A to B is the time a network packet takes to travel from A to B.

One-way delay can be challenging to measure. For this reason, the *Round-Time-Trip delay* (RTT) is often used. RTT is the time a network packet takes to travel from A to B and then back to A. It is sometimes referred to as the "ping", from the well-known Unix utility `ping`.

Section 1.5.1 explores some properties of delay, which apply both to one-way delay and RTT.

1.5.1 Delay properties

Figure 3 shows RTT samples obtained with the implementation described in Section 3, through a tunnel from Tokyo to Marseille.

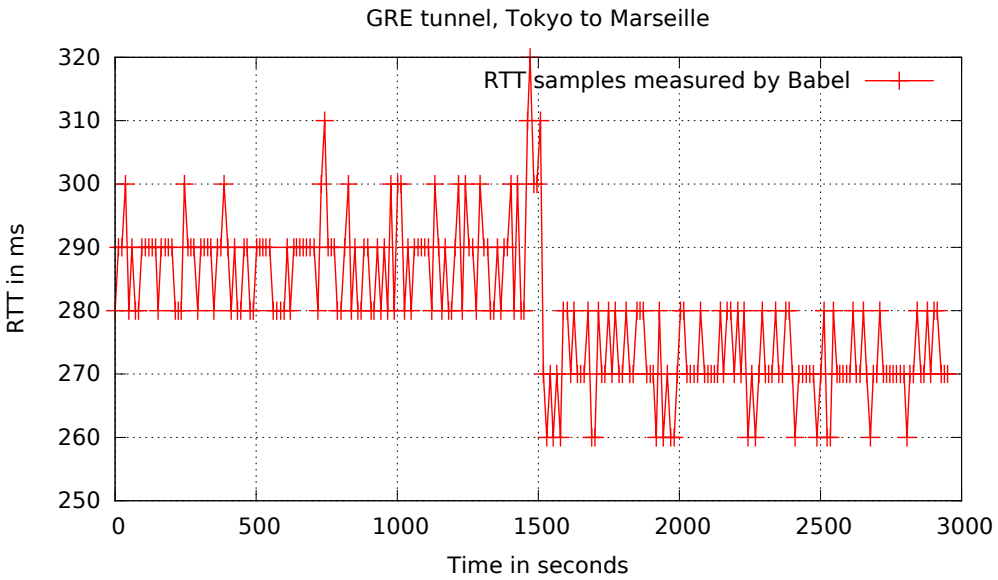

Figure 3: RTT through a tunnel. A sharp decrease occurs at time 1500, while there is a more or less constant jitter.

In this example, the RTT is the sum of two components. The first one evolves at a large timescale, with a sharp RTT decrease around time 1500. The second

one evolves very rapidly, but the variation has a small amplitude. This second component is rather a noise term.

In the general case, end-to-end delay is the sum of two components [15]:

1. a fixed component, dependent on the actual physical path between the two hosts. It includes transmission delay, physical propagation delay, and processing delay. This component is often called the *base delay* [15]. It is actually expected to vary, but at a very slow timescale (from tens of minutes to hours) because of routing changes on the underlying network. This effect is visible on Figure 3, around time 1500.

2. a variable component, due to router congestion. We call this component the *queuing delay*. This second component is a source of jitter, and can only increase the delay from its base value.

2 Using RTT as a metric

This work focuses on overlay networks. We use the RTT to build a new metric for the Babel routing protocol. We claim that this new metric is well adapted to overlay networks, where delay is a key factor.

2.1 Motivation

In an overlay network, estimating the quality of a given link can be challenging. Wireless or wired interfaces, for instance, have properties exported by the operating system — radio channel, Signal-to-Noise Ratio (SNR) or link speed — that may be used for link quality estimation. Tunnels have no such directly visible properties.

In practice, the actual path to reach the tunnel endpoint — through the Internet — may range from a very good path with a few hops to a round-the-earth path that includes a satellite link. One solution could be a manual configuration of the cost of each tunnel, but this process would be tedious and error-prone.

A more satisfying solution is to derive a cost from the *delay* on the tunnel. Indeed, a higher delay indicates a longer underlying path, which is likely to be of lesser quality.

Before this work, Babel was able to run on overlay networks, but it would consider all links to be equivalent, which is not satisfying in most cases. With the new metric, Babel is now more adapted to this specific environment.

2.2 Example situation

Consider the network depicted on Figure 4, on the left. A is sending data to D directly, since both nodes are neighbours.

When the link between A and D breaks down, the routing protocol has a choice: either it will route packets through B, or through C. The diagram on the right

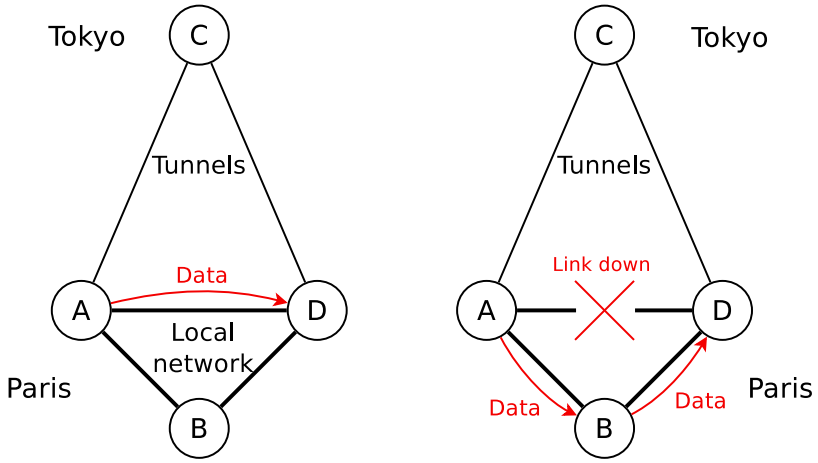

Figure 4: An example network with tunnels. When a link breaks, the new route avoids the long tunnels connecting C.

exhibits the desired behaviour: choose B and avoid the long tunnels going to and from C.

2.3 Design goals for the RTT-based metric

The following requirements were imposed for the measure of RTT, mostly because Babel already meets them.

2.3.1 Lightweight

A dynamic routing algorithm generates network traffic to disseminate routing information between nodes. In this regard, the Babel protocol is economic: studies show that it has low overhead compared to other routing protocols designed for the same type of networks [2, 14]. It is therefore important to keep control traffic as low as possible.

2.3.2 Asynchronous

`babeld` has a sophisticated scheduling scheme for outgoing messages. It means that outgoing messages are not sent immediately: they can be delayed for up to a few seconds. This is done to avoid synchronisation issues between nodes [6]. It also allows the aggregation of multiple messages into a single UDP packet, which helps lowering overhead.

In our case, it means that we cannot control at what time we send or receive messages between nodes. This calls for an asynchronous algorithm, and rules out a simple request-response protocol, such as the one used by the `ping` utility.

2.3.3 Lack of clock synchronisation

Measuring a RTT implies taking time measurements, which requires each node to have a clock.

Clock synchronisation of all nodes is often hard to ensure, even when using the Network Time Protocol (NTP). Furthermore, NTP needs a working network connection, which might not exist before running Babel — since Babel is precisely a means to obtain network connectivity.

Therefore, the algorithm must work without clock synchronisation: each node may have its own time reference.

2.3.4 Using RTT instead of one-way delay

A tunnel over the Internet is naturally an asymmetric link. Most network operators use a strategy called "hot-potato routing", which implies that the forward route is often different from the backward route.

Thus, one-way delay is expected to provide a more accurate estimation of the cost than round-time-trip delay. Indeed, experiments with delay-sensitive congestion control have shown that one-way delay behaves sensibly better than round-time-trip [9, 15].

Unfortunately, the lack of clock synchronisation makes one-way delay hard to measure. TCP-LP [9] and LEDBAT [15] both measure one-way delay, but are only interested in the queuing delay (*i.e.* the variable component). This can be computed by getting rid of the constant component, with includes the clock offset. In our case, we are interested in the constant component, and we cannot get rid of the clock offset in the same way.

For the sake of simplicity, we chose to base our work on round-time-trip delay. Since we need only a rough estimation of the quality of a tunnel, we feel that the refinement provided by one-way delay is not worth the added complexity.

2.4 Algorithm for asynchronous RTT measurement

This section describes the algorithm used to measure the RTT between two neighbouring nodes, *i.e.* nodes that can directly talk to each other over a given link.

The algorithm is inspired by Mills' HELLO protocol [11]. It allows a node to measure the RTT to each one of its neighbours.

2.4.1 Description

Figure 5 shows a *sequence diagram* of the communication involved. On this diagram, time flows from top to bottom. The vertical arrows show the history of each node, and dotted arrows show a message travelling from one node to the other.

A sends a message to B, recording the time t_1 of emission — according to A's local clock. Upon receiving the message, B also records the time t'_1 of reception,

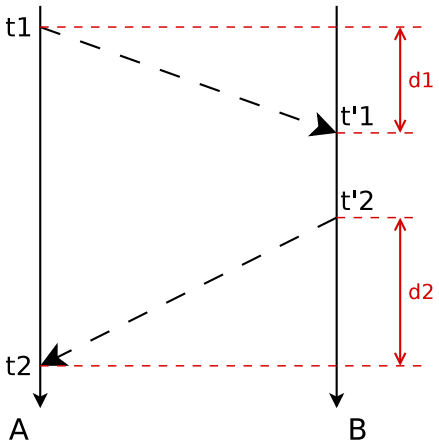

Figure 5: Sequence diagram of an asynchronous measurement of RTT, initiated by A. The RTT is $(t_2 - t_1) - (t'_2 - t'_1)$.

also according to its own local clock. B can then reply in the same way anytime it likes, here at time t'_2.

With the convention used on Figure 5, the RTT is then calculated by A as

$$RTT_A = d_1 + d_2 = (t_2 - t_1) - (t'_2 - t'_1) \tag{1}$$

2.4.2 Properties

The key idea is that the two nodes' clocks *do not need to be synchronised*, which is a requirement described in Section 2.3.3. The non-synchronisation of the two clocks means that we cannot compare t_i and t'_j, since they are absolute timings taken from different clocks.

Consider the two terms of the equation, $(t_2 - t_1)$ and $(t'_2 - t'_1)$. t_1 and t_2 are taken from the same clock, so that they can be compared. $t_2 - t_1$, then, is a *time interval* (i.e. duration), which is a relative value. The same goes for t'_1 and t'_2. We then substract two durations, which is a valid operation.

The only assumption is that both clocks tick roughly at the same speed during the measurement. In our implementation, the measurement takes a few seconds, which is sufficiently low to fulfil this assumption, even with cheap hardware.

2.5 Noise filtering

As discussed in Section 1.5.1, the delay has two components. In our case, we are interested in the *base delay* and want to filter out the other, noisy, component.

Unfortunately, we have no direct access to the base delay.

Since the queuing delay only *increases* the RTT from its base value, an idea would be to take the minimum of RTT samples. Since we expect the base RTT to

vary, we need a sliding window of time. The length of this window should be set to the expect time of the variations of the base RTT.

However, this method has three drawbacks:

1. it requires keeping samples from the whole window in memory, which might not be suitable for embedded platforms;
2. tuning the window length is not easy, as we want a quick response when the base RTT changes, while being immune to noise;
3. because of stability issues, we would like the base RTT to evolve smoothly, and not to "jump" instantly from one value to another. Using a running minimum may exhibit this undesired behaviour.

Instead, we use an exponentially weighted moving average:

$$s_0 = x_0 \tag{2}$$

$$s_n = \alpha \cdot x_n + (1 - \alpha) \cdot s_{n-1} \tag{3}$$

where x_i are RTT samples, s_i are smoothed values, and $\alpha \in]0; 1[$ is the configurable smoothing factor.

This method has the following advantages:

- no need to keep samples into memory;
- the noise is "smoothed out";
- we still converge quickly enough when the base RTT changes.

Of course, α needs to be tuned to get a satisfying compromise between the last two points. The effect of this smoothing method is shown on Figure 6.

2.6 Computing a cost from the RTT

Once we have an estimation of the base RTT, we need to use it to compute a cost, as the RTT itself is not directly usable. Our implementation uses the function shown on Figure 7, which is added to the normal cost used by `babeld`.

For low RTT, no cost is added, since our implementation is not accurate below 10 ms. For medium RTT, the cost is affine in the RTT. For high RTT, the additional cost has a upper limit. We firmly believe that in the general case, the function must be bounded in order to achieve some form of stability. The idea is that at high RTT, all links should be treated as equally bad, even if some are worse than others: this avoids switching needlessly if all links have high RTT. The results shown on figures 12 and 13 seem to confirm this hypothesis; see also Section 4. However, we do not have any theoretical result yet.

Note that this function is a purely local matter: in the same network, different nodes can use different functions without any issue.

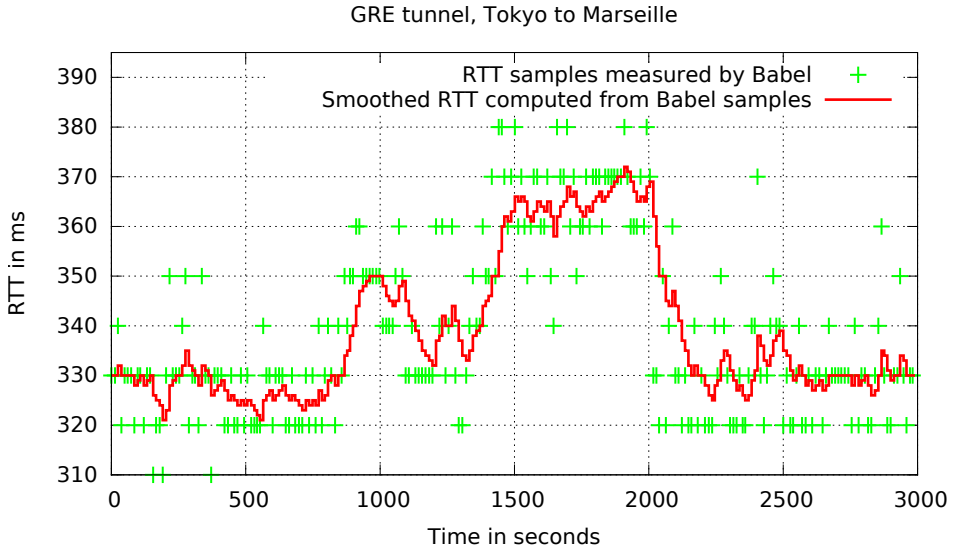

Figure 6: Effect of the smoothing algorithm on a real tunnel, with $\alpha = \frac{42}{256} \simeq 0.164$.

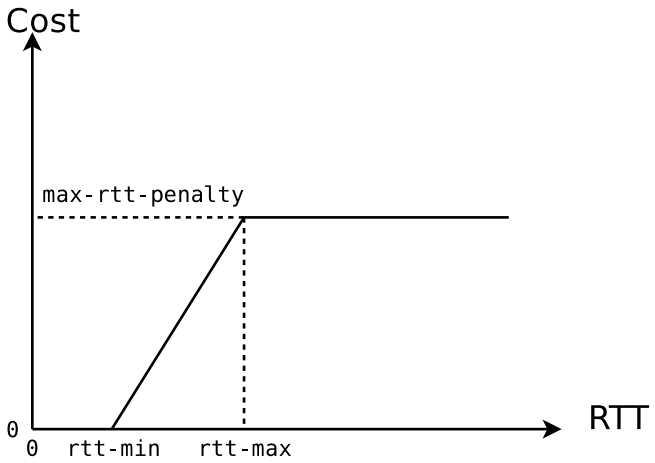

Figure 7: Cost as a function of RTT. Note that this cost is *added* to the existing cost (*e.g.* 96 for a wired interface).

2.7 Previous work

Historical usage In 1983, Mills described the use of RTT for routing in the DCNet [11]. He does not, however, provide any evaluation of his implementation.

Later, around 1989, RTT was used in the ARPANet to optimise routing [8]. The network had a very limited capacity, and it was experiencing increasing contention. By taking RTT into account, the network architects managed to partially alleviate this contention.

The asynchronous algorithm we use to measure RTT (described in Section 2.4) is inspired from Mills' work [11]; it later became the basis for NTP [12].

Using delay-based metric for routing has been seemingly abandoned since then: to the best of our knowledge, no modern network has been using this method for years. Our interpretation is that the advent of very fast core networks has deprecated this technique. Indeed, contention is now rarely experienced inside of the core network itself, which deprecates the use of clever ways to fight it off. Instead, we believe that contention has been deported to the edge of the network (*i.e.* the Customer Premises Equipment (CPE) of the end-user, often connected with an asymmetric and low-capacity technology such as xDSL), which typically doesn't use a routing protocol at all.

Modern routing protocols IGRP and EIGRP, two modern routing protocols from Cisco, also use a parameter called "delay" for computing a metric[7]. However, it is a value configured by hand, and is not the result of a measurement. It is therefore unrelated to our work.

3 Implementation

This section covers the implementation of a RTT-based metric in Babel. The protocol extension is described in greater detail in [7].

3.1 Protocol extension

TLVs and sub-TLVs In the Babel protocol, a given message is encoded as a (`type`, `length`, `value`) 3-uple, called a TLV [3]. The `type` describes which kind of message is sent out, while the actual content of the message (of length `length`) is sent out as `value`.

An interesting property is the ability to fit additional data inside a TLV, which is commonly referred to as *piggybacking*. All TLVs are *self-terminating*, in the sense that their actual length (the "base length") can be determined without reference to the explicit `length`. If the explicit `length` of a TLV is larger than its base length, the extra space present in the TLV is silently ignored by an implementation of

[7] See `http://www.cisco.com/en/US/tech/tk365/technologies_white_paper09186a0080094cb7.shtml#metrics`, retrieved 11th July 2013.

the base protocol, and is used by extended implementations to store a sequence of *sub-TLVs* [4].

Our extension The algorithm described in section 2.4 requires node A to know node B's timing measurements, namely t'_1 and t'_2.

To keep a low overhead, the adopted solution is to *piggyback* the timing measurements into existing Babel TLVs, using sub-TLVs [4]. The Babel protocol defines two types of messages that are interesting for our work, since they are exchanged every few seconds between neighbours:

Hello message, which is sent at a regular interval to all neighbours;

IHU (I Heard You) message, which is used to confirm that a Hello message was received.

Using the sub-TLV format described in Appendix A, this allows taking a RTT sample every few seconds, assuming no packet loss.

3.2 Code

My implementation effort was based on `babeld`, the reference implementation of the Babel protocol. `babeld` currently consists of 8411 lines of C code (as of 19th August 2013).

The implementation consists of 18 commits, with 11 files changed, 389 lines added, and 19 lines removed[8]. This brings the total amount of C code to 8680 lines in `babeld`.

The modifications may be broken down into three rough sets:

Message parsing Since the extension defines a new sub-TLV type, I extended `babeld`'s packet parser. Note that sub-TLVs are a relatively new feature of Babel, and there is no generic sub-TLV parsing code in `babeld`.

Timestamping outgoing packets To get accurate RTT measurements, we need to timestamp messages at the time they are sent on the wire. The most accurate would be a hardware timestamp, but cannot be implemented easily. Instead, we timestamp in software, as late as possible — just before passing the messages to the kernel for sending.

Unfortunately, `babeld` has a sophisticated scheduling scheme for outgoing messages, as described in Section 2.3.2, which means that messages may be buffered for a few seconds. Since changing `babeld`'s scheduling is not desirable, I had to create messages with an empty timestamp, and plug some code to fill the timestamps just before sending the messages.

[8] This includes documentation and comments.

Computing a cost From the RTT samples, we compute a smoothed RTT for each neighbour. The smoothing factor is configurable.

To obtain a cost, the function shown on Figure 7, Section 2.6, is encoded as three parameters: `rtt-min` and `rtt-max`, in milliseconds, and `max-rtt-penalty`.

These parameters can be set individually for each interface, which allows using the extension on some interfaces only. This is useful when running Babel with a mix of wireless interfaces and tunnels.

4 Stability

Figure 11 shows a situation where a stability issue can arise. Indeed, while A is using one of the link at full utilisation, the RTT on the link will increase, and so will the metric of the considered route. A will eventually switch and use the other link. But then, the other link will also see its RTT increase, while the first link is mostly idle. A will eventually switch back and forth between the two links, which forms a stability issue.

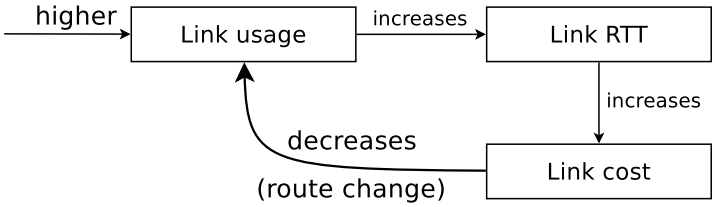

Figure 8: Negative feedback loop. Case of an initial increase in link usage.

More formally, the interaction can be represented as a feedback loop, as shown on Figure 8.

Feedback loop Note that we have a *negative* feedback loop, which should stabilise things. However, in practice, we have an unstable behaviour (see Section 5.2).

This can be explained in the following way. The negative feedback loop makes the routes converge to an equilibrium point. However, the available configuration space (which route to chose) is *discrete*. Thus, the equilibrium point may not always be reached, and the system will oscillate between points that are close to the equilibrium point.

4.1 Preventing stability issues

According to practical measurements on real tunnels, the RTT does depend on link usage but in a complicated way. An example is given in Appendix C.

Our implementation makes some efforts to reduce the effect of a potential feedback loop. The techniques used are:

- smoothing of the RTT samples, as discussed in Section 2.5;

- non-linear behaviour of the "metric = f(RTT)" function. The positive effect of this is discussed in Section 5;
- smoothing of the metric (hysteresis). This is already a part of `babeld` since the 1.4 release.

5 Evaluation

This section details the various experiments conducted in order to validate the implementation. See appendix B for the underlying methodology, especially regarding measurements.

We are grateful to Jean-Paul Smetz, Julien Muchembled and Cédric de Saint Martin, from Nexedi, for providing the infrastructure used for the tests. It consists of virtual machines located in various places around the world.

5.1 Real-world behaviour

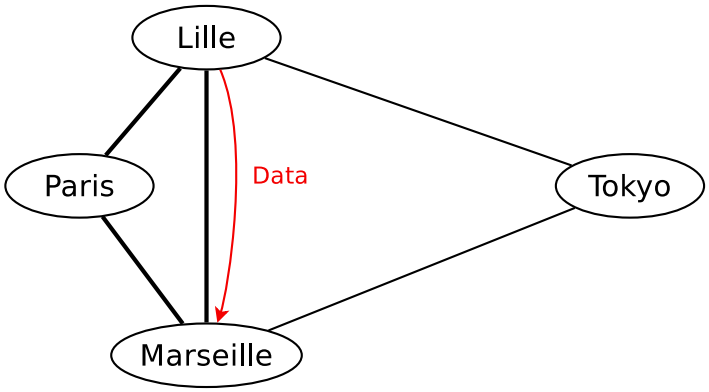

Figure 9: Real-world setup to test the RTT metric.

The first experiment is a real-world setup of the situation described in Figure 4. The actual setup is depicted in Figure 9, and has the following properties:

- each node is a virtual machine running Debian;
- the tunnels are made with OpenVPN over UDP, without cryptography;
- all `babeld` instances run with the following parameters: `rtt-min` = 10ms, `rtt-max` = 200ms, `max-rtt-penalty` = 150.
- Lille is sending data to Marseille, using `iperf` in UDP mode;
- at t_1, the Lille $\leftrightarrow$ Marseille link is shut down. At t_2, the Paris $\leftrightarrow$ Marseille link is shut down. At t_3, both links are simultaneously turned on again.

The graph at Figure 10 shows the incoming traffic for Marseille, on each of its three interfaces. We can see that each time a link goes down, the sender switches route after only a few seconds. Moreover, it avoids taking the long tunnel through Tokyo, unless there is no other route available. Conversely, when the direct link comes back, the traffic switches back to it.

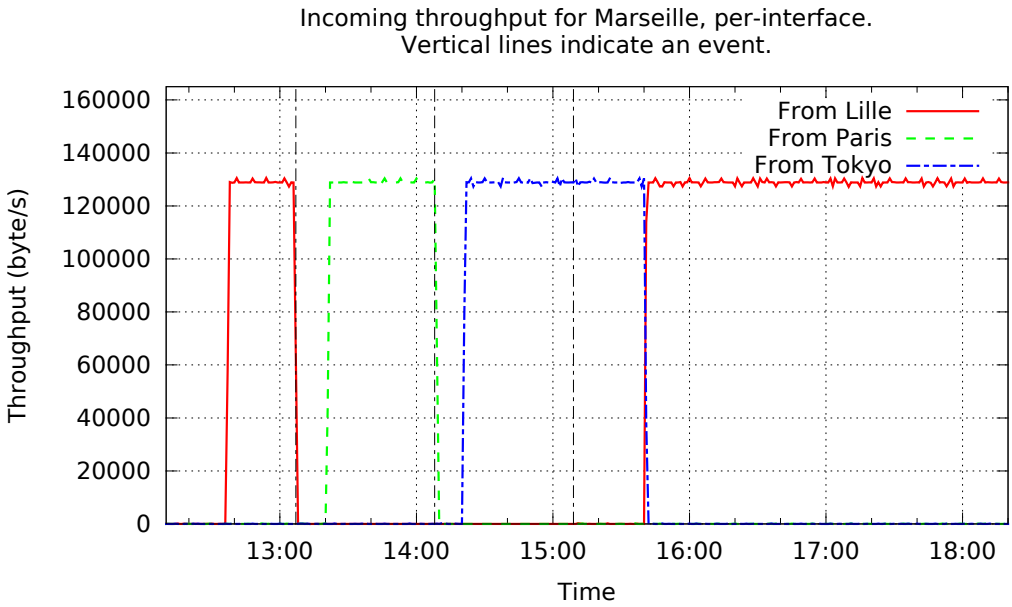

Figure 10: Real-world behaviour: incoming throughput for Marseille, while receiving iperf traffic from Lille.

Figure 10 in Appendix C gives an alternative vision of the same experiment.

5.2 Stability

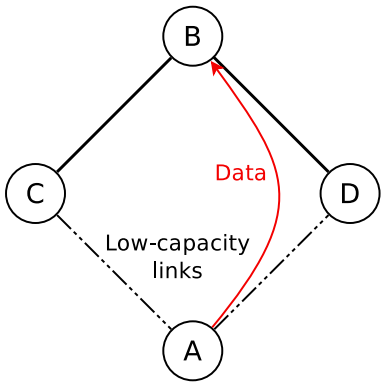

Figure 11: Simulated network to exhibit stability issues.

As discussed in Section 4 (see in particular Figure 8), a feedback loop can lead to an unstable behaviour.

Since the feedback loop may still occurs in some cases, I simulated a network with extreme characteristics, which exhibits the feedback loop. The simulation was performed with containers (lightweight virtual machines), which allows running the `babeld` daemon on each container as usual. The exact methodology is described in Appendix B.

As shown on Figure 11, a node A has two possible routes to reach node B, both of which make use of a low-capacity link.

We expect this situation to exhibit an oscillating behaviour. The result of the simulation, shown on Figure 12, confirms this intuitive reasoning.

In the simulation, the low-capacity links are emulated thanks to the `netem` queuing discipline, used on A's outgoing interfaces. A is using `iperf` in TCP mode to send data to B.

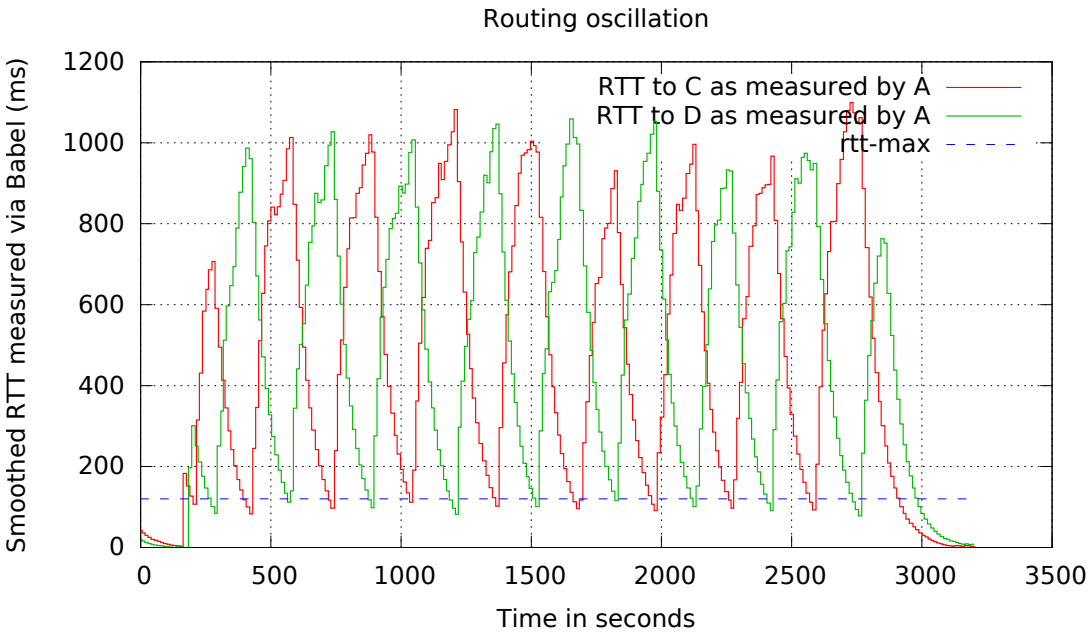

Figure 12: RTT oscillation for the simulated network of Figure 11, with `rtt-min` = 10ms, `rtt-max` = 120ms, and `max-rtt-penalty` = 150.

Figure 12 shows the RTT from A to its neighbours during the 45-minutes `iperf` session. As expected, a routing oscillation occurs. However, the period of the oscillation is 300 seconds (measured thanks to the throughput log), which is acceptable.

One of the reasons for such a low frequency (which means stability) is the saturating behaviour of the function f — computing the cost of the link from the RTT. Most of the RTT values are way above `rtt-max`, which helps to avoid

switching too often. The intuitive behaviour is the following: when both links are obviously overloaded (*i.e.* a RTT above `rtt-max`), we won't gain much by switching anyway.

Figure 13 shows the same situation with a very large `rtt-max` and a `max-rtt-penalty` adjusted to obtain the same slope for f[9], effectively removing the saturating behaviour. The new oscillation period is 50 seconds (again measured thanks to the throughput log), which is still acceptable but much less satisfactory.

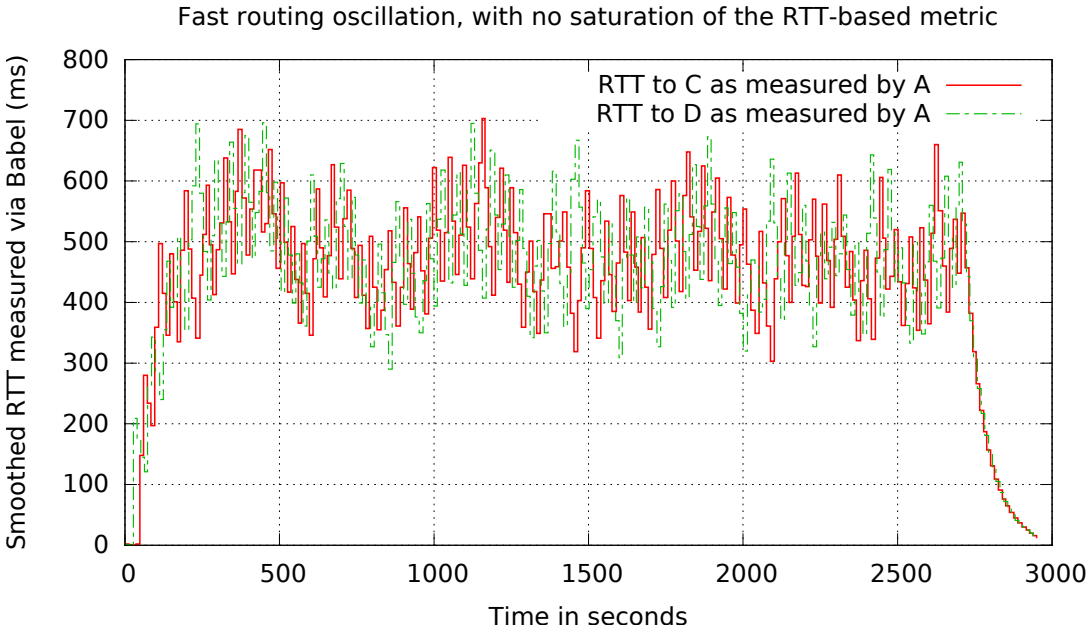

Figure 13: RTT oscillation for the simulated network of Figure 11, with no bound on `rtt-max`.

6 Conclusion

We have shown that a delay-based metric is useful in the presence of tunnels. Additionally, it can be implemented in a fully asynchronous way, without intrusive changes to a routing protocol such as Babel and with little overhead.

Furthermore, with the proper setup (smoothing and non-linear behaviour), potential stability issues can be overcome.

6.1 Contributions

Protocol development Together with Juliusz Chroboczek, we developed the asynchronous protocol used to measure RTT, adapted from the work of David L.

[9] `rtt-max` = 1990ms and `max-rtt-penalty` = 2700.

Mills. It is quite generic and should be applicable to other routing protocols, such as OSPF or EIGRP.

Code My work led me to extend `babeld`, the reference implementation of the Babel routing protocol. My code has been reviewed and accepted, and will hopefully soon be available as part of version 1.5 of `babeld`. See Section 3.2 for more details.

Documentation Since it required an extension to the Babel protocol, Juliusz Chroboczek and myself are in the process of writing an IETF Internet draft [7] describing the extension, in accordance with the Babel extension specification [4].

Evaluation Thanks to a real overlay network, I was able to verify the validity of my implementation. I also evaluated the same implementation in a simulated environment, in order to experiment with less favorable cases. While my implementation may exhibit an oscillating behaviour in such cases, it is still acceptable, thanks to the techniques used to prevent oscillation.

Public exposure I organised a public workshop on Babel at Pas Sage en Seine[10] in June. In July, I co-organised a three-days Babel "hackathon"[11], dedicated to presenting and testing the new features of Babel (which includes my work).

6.2 Further work

It would have been interesting to port my extension mechanism to the Quagga version of `babeld`[12], in order to emphasise the compatibility between different implementations of the protocol. However, I haven't had the time to do so.

A thorough theoretical study of stability issues would be highly appreciated, since we mostly rely on intuitions and experimentation at present.

My implementation is based on the 1.4 branch of `babeld`, which includes a "smoothed metric" algorithm supposed to reduce oscillations. Since this algorithm has not yet be thoroughly tested in real networks, it would be interesting to see if it played a decisive role on my experiments.

[10] `http://www.passageenseine.org/Passage/pses-2013/pas-sage-en-seine-2013#s1214w`, retrieved 18th August 2013.

[11] `http://leloop.org/2013/07/13/hackathon-babel/`, retrieved 18th August 2013.

[12] See `http://www.pps.univ-paris-diderot.fr/~jch/software/babel/#download`, retrieved 18th August 2013.

7 Bibliography

References

[1] Ulysse Beaugnon, "Building a resilient overlay network: Re6stnet", *Internship at Nexedi KK*, September 2012. `http://community.slapos.org/P-ViFiB-Resilient.Overlay.Network/Base_download`

[2] Mehran Abolhasan, Brett Hagelstein and Jerry Wang, "Real-world performance of current proactive multi-hop mesh protocols", *Asia Pacific Conference on Communications (APCC 2009)*, 2009.

[3] Juliusz Chroboczek, *The Babel Routing Protocol*. RFC 6126, February 2011.

[4] Juliusz Chroboczek, *Babel Extension Mechanism*. Internet draft `draft-chroboczek-babel-extension-mechanism-00`, June 2013. Available from `https://tools.ietf.org/html/draft-chroboczek-babel-extension-mechanism-00`.

[5] R. Coltun, D. Ferguson, J. Moy, A. Lindem *OSPF for IPv6*. RFC 5340, July 2008.

[6] Sally Floyd and Van Jacobson, "The synchronization of periodic routing messages", *IEEE/ACM Transactions on Networking*, **2**:2, pp. 122-136, April 1994.

[7] Baptiste Jonglez and Juliusz Chroboczek, *Delay-based Metric Extension for the Babel Routing Protocol*. Internet draft, 2013. Available from `http://www.pps.univ-paris-diderot.fr/~jch/software/babel/draft-jonglez-babel-rtt-extension.txt` (work in progress)

[8] Atul Khanna and John Zinky, "The Revised ARPANET Routing Metric", *SIGCOMM '89 Symposium proceedings on Communications architectures and protocols*, pp. 45-56. ACM New York, NY, USA, 1989.

[9] Aleksandar Kuzmanovic and Edward W. Knightly, "TCP-LP: low-priority service via end-point congestion control", *IEEE/ACM Transactions on Networking*, **14**:4, pp. 739-752, August 2006.

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

[11] David L. Mills, *DCN Local-Networks Protocols*. RFC 891, 1983.

[12] David L. Mills, J. Martin, Ed., J. Burbank and W. Kasch, *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905, June 2010.

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

[14] David Murray, Michael Dixon and Terry Koziniec, "An Experimental Comparison of Routing Protocols in Multi Hop Ad Hoc Networks", *Australasian Telecommunication Networks and Applications Conference (ATNAC 2010)*, 2010.

[15] S. Shalunov, G. Hazel, J. Iyengar and M. Kuehlewind, *Low Extra Delay Background Transport (LEDBAT)*. RFC 6817, December 2012.

[16] João Luís Sobrinho, "Algebra and Algorithms for QoS Path Computation and Hop-by-Hop Routing in the Internet", *Proceedings of the IEEE INFOCOM 2001*, pp. 727-735, Anchorage, Alaska, April 2001.

A Packet format

This appendix describes the packet format used by our extension. Note that this document is not an authoritative source, as the packet format may change after the publication of this report. The extension draft [7] should be consulted for an up-to-date specification.

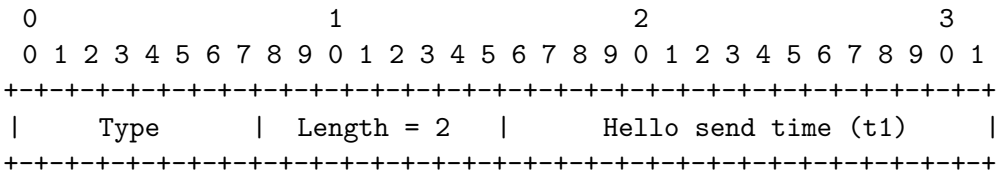

Figure 14: Sub-TLV included in Hello messages, using the convention of Figure 5.

The sub-TLV included in Hello messages is shown on Figure 14. Sent by A (with the convention used on Figure 5), it contains the time at which the message was sent, encoded as an integer, in centiseconds modulo 2^{16}.

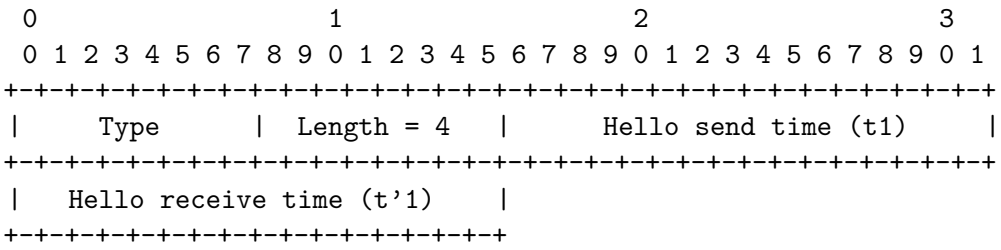

Figure 15: Sub-TLV included in IHU messages, using the convention of Figure 5.

The sub-TLV included in IHU messages, shown on Figure 15, contains two timing measurements. It is sent from B to A, as a follow-up to a previous extended Hello message.

The first timing measurement, "Hello send time", is a copy of the value found in the latest extended Hello message from A. This avoids A to remember which extended Hello message it has sent. The second timing measurement, "Hello receive time", is the time at which B received the extended Hello message, according to its local clock.

Upon receiving both an extended Hello and extended IHU messages from B, A is able to compute a RTT sample. This allows taking a RTT sample every few seconds, assuming no packet loss.

B Evaluation methodology

This appendix describes the methodology used for the evaluation, both for the simulations and the real-world situations.

B.1 Instrumentation

Whether running a simulation or using a real network, I needed to measure some data. This includes:

- information gathered from `babeld`, such as RTT samples, smoothed RTT and metrics. They were measured either from the local interface[13] or by parsing `babeld`'s debug output, in order of decreasing preference;
- information gathered on the system, such as network throughput or RTT as seen by `ping`.

All data was stored in plain text files. Each line is a sample of the data, with the first column being the time and the subsequent columns containing the data of the sample itself. `gnuplot` was later used to exploit the raw data.

Handling of time For a given experiment, I often needed to consider multiple data source simultaneously (*e.g.* looking at the evolution of the RTT when the throughput is changing). To be able to compare data from multiple sources, a common time reference was necessary.

I chose to use Unix timestamps[14] for all measurements, as measured by `gettimeofday(3)`. One reason for this choice is `gnuplot`'s built-in support for Unix timestamps.

This method assumes that the clock remains stable during a given experiment. This is achieved by synchronising the clock with NTP beforehand.

RTT samples RTT samples are only available from `babeld` debug output. To record them, I piped `babeld`'s output through the following shell script.

```
# -u : unbuffered mode (for accurate timestamps)
sed -u -e 's/^RTT to \(.*\) on \(.*\) sample result: \(.*\) \
           ms./\1 \2 \3/' | while read neighbour iface rtt
do
    [ "$neighbour" = "$NEIGHBOUR" ] && echo "$(date +%s.%N) $rtt"
done
```

[13] A local socket on which `babeld` provides some information about neighbours and routes.

[14] Number of seconds since January 1, 1970

Smoothed RTT and metrics For a given neighbour, we may want to keep track of its (smoothed) RTT. For a given route, we may want to know its current metric. To record both of these data, I wrote a small python program that connects to `babeld`'s local interface and parses the data sent there.

Network throughput On Linux, the `/proc/net/dev` file keeps a count of the total number of bytes sent or received on each interface. I wrote a python script that periodically parses this file and estimates the current throughput, by computing differences of the bytes count.

The basic algorithm is described below, in Python syntax.

```
now = time.time()
rxbytes, txbytes = parse("/proc/net/dev", interface)
print(‘‘Timestamp RX TX’’)
while True:
    time.sleep(interval)
    now_prev = now
    now = time.time()
    rxbytes_prev, txbytes_prev = rxbytes, txbytes
    rxbytes, txbytes = parse("/proc/net/dev", interface)
    rx = (rxbytes - rxbytes_prev) / (now - now_prev)
    tx = (txbytes - txbytes_prev) / (now - now_prev)
    print((now + now_prev) / 2, rx, tx)
```

B.2 Simulation

In order to simulate a network with various properties, I used a container technology, LXC, which is based on "isolation" support from the Linux kernel.

Lightweight virtualisation LXC allows to run multiple instances of the entire network stack (network interfaces, routing table...) within the same kernel. Each *container* has its own virtual interfaces and routing table, and can be connected to any other container using virtual links. A separate `babeld` process is run on each container.

This solution is more lightweight and convenient than a full-blown virtualisation technology. For instance, all containers share the same filesystem, which allows all the containers to share the same `babeld` binary.

Inside the container, the network interfaces are `veth` interfaces. To connect two or more containers together, a bridge is used on the host.

Traffic control I needed to simulate links with a given delay or capacity, to emulate various kind of tunnels. The Linux kernel has all facilities necessary for traffic control, available through the `tc` command. I used the `netem`[15] queuing

[15] `netem` stands for "Network Emulator".

discipline, as it is simple to use and allows to set both a delay and a maximum throughput on a given link.

To add delay to traffic going out of an interface: `tc qdisc add dev eth0 root netem delay 1000ms`. To specify a maximum outgoing throughput: `tc qdisc add dev eth0 root netem rate 100kbps`.

C Other evaluation data on stability

This section provides additional figures and data relating to the evaluation described in Section 5.

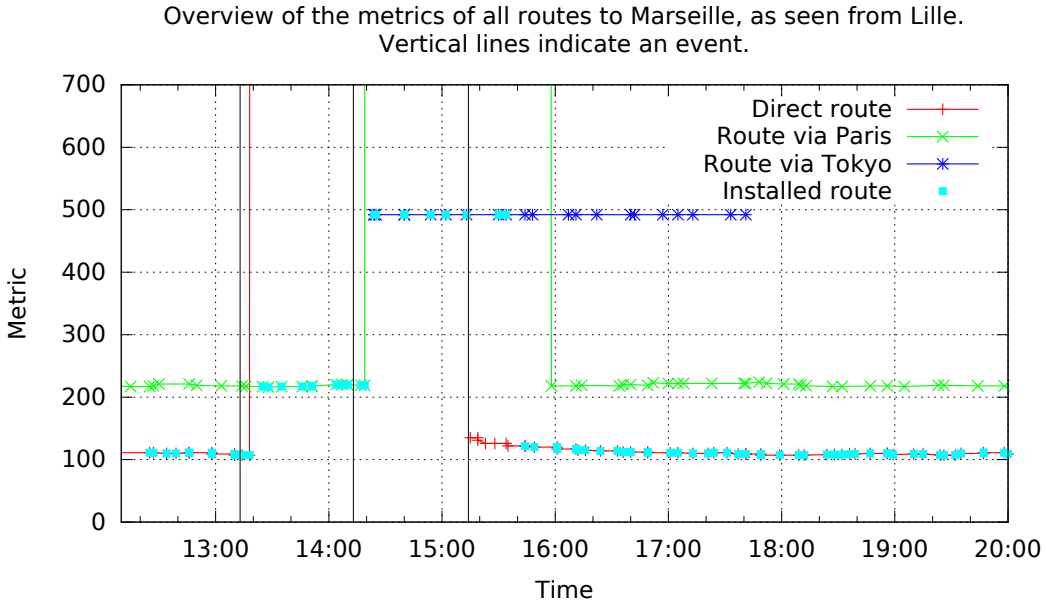

Figure 16: Real-world behaviour: evolution of metrics for the routes from Lille to Marseille.

Figure 16 refers to the same experiment as Figure 10. It shows the metric over time of the three routes leading to Marseille, as seen from Lille. The blue points represent the metric of the installed route. Since the RTT between the French nodes and Tokyo is very high (roughly 320 ms), we see that the route through Tokyo takes twice the maximum penalty, *i.e.* $2 \times$ `max-rtt-penalty` $= 300$.

RTT as a function of link usage Figure 17 shows the RTT through an ADSL link. During the measurement, TCP iperf sessions are periodically performed through the tunnel to saturate the ADSL uplink.

In this case, the RTT increases about fourfold when the link is heavily loaded.

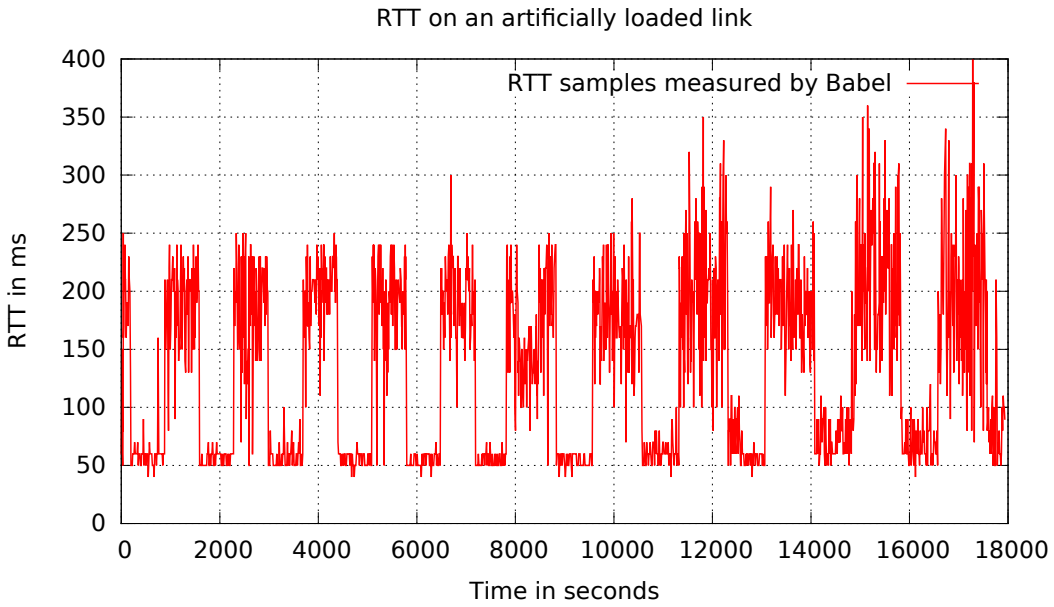

Figure 17: RTT through a tunnel supported by an ADSL link, while periodically loading the link.