Distributed and Adaptive Routing Based on Game Theory

Baptiste JONGLEZ and Bruno GAUJAL
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

***Abstract*—In this paper, we present a new adaptive multi-flow routing algorithm to select end-to-end paths in packet-switched networks. This algorithm provides provable optimality guarantees in the following game theoretic sense: The network configuration converges to a configuration arbitrarily close to a pure Nash equilibrium. In this context, a Nash equilibrium is a configuration in which no flow can improve its end-to-end delay by changing its network path. This algorithm has several robustness properties making it suitable for real-life usage: it is robust to measurement errors, outdated information, and clocks desynchronization. Furthermore, it is only based on local information and only takes local decisions, making it suitable for a distributed implementation. Our SDN-based proof-of-concept is built as an Openflow controller. We set up an emulation platform based on Mininet to test the behavior of our proof-of-concept implementation in several scenarios. Although real-world conditions do not conform exactly to the theoretical model, all experiments exhibit satisfying behavior, in accordance with the theoretical predictions.**

I. INTRODUCTION

A common challenge in current and future communication networks is to exploit path diversity to increase performance and/or reliability. This problem can be modeled as a *multi-commodity flow problem* [1]. Given a number of concurrent source-destination flows, the problem is to assign these flows to network paths, while respecting capacity constraints and optimizing a performance metric. Our goal is to provide a *distributed* solution to this problem, that requires neither cooperation between sites nor knowledge of the network topology and performance parameters. We restrict ourselves to the *non-splittable* case, in which a given source-destination flow uses only one path at a given time. This is necessary in practice to avoid persistent packet reordering, which would harm performance. However, a given flow is allowed to change its choice of path several times during its lifetime.

Related work: When optimizing for a single flow with no *a priori* knowledge about the network, a multi-armed bandit model [2], [3] (or more generally any other kind of learning automata) can be used for learning the shortest-delay path. However, since we consider multiple competing flows, we need a more complex model based on game theory.

Regarding the multi-flow problem, convergence results for a wide class of online decision algorithms have been proven in [4], but in a different setting. The authors consider non-atomic routing games, *i.e.* each packet in a given flow independently chooses a path. This setting is hard to apply in a practical network because of packet reordering.

On the practical side, most existing approaches for exploiting multiple paths in a network are either limited to very specific settings, *e.g.* ECMP (Equal-Cost Multi-Path forwarding) for parallel paths with similar characteristics, or targeted at datacenter networks [5], [6]. As such, they typically assume a network with a very organized and hierarchical topology, for instance fat-tree, or require explicit information sharing from the network [5]. We make no such assumptions.

When working with end-hosts directly, Multipath TCP [7] allows using multiple paths for a single source-destination flow, improving performance and reliability. This is the most promising approach so far, but requires extensive modifications in the operating system of both ends of the communication, which hampers large-scale deployment for now. In contrast, our proof-of-concept only requires end-hosts to include TCP timestamps or use a LEDBAT-based transport protocol, which is already widespread [8].

A noteworthy class of routing algorithms bases its routing decisions on dynamic properties of the network, including real time link load, end-to-end latency, or packet loss. This class of strategies is known as *adaptive routing*. Such solutions are attractive because they typically enable more efficient usage of network resources. In contrast, traditional load-oblivious routing algorithm may blindly over-utilize some links, leading to congestion, while other links have unused capacity. However, despite years of research, adaptive routing techniques did not see wide adoption. The main reason is the presence of potential instabilities and routing oscillations, which could make the cure worse than the disease. Indeed, early experiments with delay-based routing in the ARPANET resulted in severe stability issues under high load, rendering the network close to unusable [9]. A possible tradeoff is to adapt routing paths at a much slower timescale. For instance, Link Weight Optimisation [10] chooses paths based on a "representative" traffic matrix, which is typically updated no more than once a day. This solves the stability issue, but it is no longer possible to adapt to real-time network conditions. More recent solutions have been proposed for adaptive routing [5], [6], [11]. Some of these solutions are based on heuristics and offer no real guarantees on their performance. Furthermore, stability issues are typically overlooked, or no convergence guarantee is provided.

Contributions: In this work, we present a novel algorithm for adaptive routing in packet-switched networks, mapping source-destination flows to paths. We find that it provides a viable and stable solution to adapt to traffic conditions,

978-0-9883-0453-6/17 $31.00 © 2017 ITC
DOI 10.1109/ITC.2017.28

and effectively avoids congestion. Our algorithm is based on strong theoretical grounds from game theory, while our proof-of-concept leverages SDN protocols to ease implementation. Our routing algorithm is endowed with the following desirable properties for efficient implementation:

- It is **fully distributed**: only local information is needed, and it requires no explicit coordination between routers;
- It is **oblivious** to the network topology;
- It is **robust** to outdated and noisy measurements;
- There are **no endless oscillations**;
- It does **not require clock synchronization** between routers, between end hosts, or between routers and end hosts.

Detailed proofs and additional material are available in a companion research report [12].

II. Distributed Routing over a Network

A. Problem description and definitions

Let (V, E) be a communication network over a set V of nodes and a set E of bi-directional links, over which we consider the following *multi-commodity flow problem*. A set $\mathcal{K}$ of *flows* of packets must be routed over the network. Each flow $k \in \mathcal{K}$ is characterized by a source a_k, a destination b_k and a nominal arrival rate of packets, λ_k. Also, each flow is assigned a set $\mathcal{P}_k$ of possible paths in the network from its source to its destination, with $|\mathcal{P}_k| = P_k$. A *configuration* is a choice of one path per flow.

Our objective is to find a configuration that minimizes *end-to-end delays* of each flow. Actually, other criteria could have been chosen, such as loss rate, round trip times, or goodput, that would require a minimal adaptation of our algorithm.

This type of question has been heavily studied in the literature in many different ways [1]. Our objective here differs from most previous work: We design a *learning algorithm* that allows each flow to discover a path, such that the global configuration is a Nash equilibrium of the system. Namely, after convergence, no flow can improve its delay by changing its path.

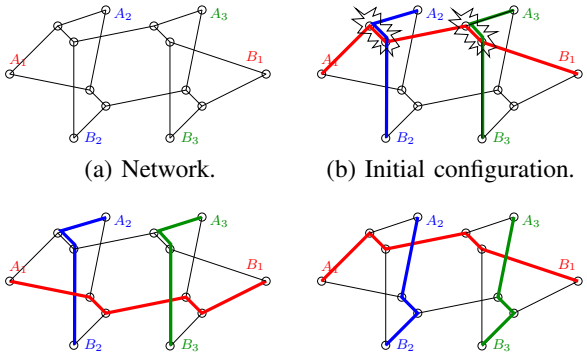
The challenge is to consider a realistic scenario in which no flow has information about the choices of the others or even knows the presence of other flows. The only information that a flow can get from the network is an estimation of the end-to-end delay of its packets sent over its current chosen path. To make things even more realistic, we assume that the delay measurements can be perturbed by random noise.

The difficulty also comes from the interdependence between the flows: when one flow decides to send its packets on a new path, this may alter the delays for the other flows because of resources sharing between the paths (either links or routers).

In this context we design a distributed algorithm that allows each flow k to eventually choose a path in $\mathcal{P}_k$ such that all alternative paths would offer a larger delay. This goal is achieved by using optimization methods coming from game theory.

An illustrative example is given in Figure 1 with a networks whose links all have the same bandwidth and three flows $(A_1, B_1), (A_2, B_2), (A_3, B_3)$.

If we start from the configuration 1.(b), the flows share some links, hence inducing large delays. The goal is to achieve one of configuration 1.(c) or 1.(d), by letting all flows explore simultaneously their paths and achieving such a coordination in a fully distributed way, in the sense that no flow has information about the presence of other flows and only has information about the delay suffered by its packets on its current route.



(a) Network. (b) Initial configuration.

(c) Desired configuration 1. (d) Desired configuration 2.

Fig. 1: Figure (a) displays a network with 3 flows $(A_1, B_1), (A_2, B_2), (A_3, B_3)$. (b) shows a configuration that suffers from congestion on two links (marked by star shapes). The final configurations (c) and (d) have no congestion: the delays for all flows have improved.

B. OPS Algorithm

Here is a description of the algorithm that each flow k runs independently. It is based on a mirror-descent learning algorithm for general potential games, presented in [13] together with a its positioning with respect to the literature.

For one flow, say k, we call $d_k(p_1, \cdots, p_k, \cdots, p_K)$, the end-to-end *average delay* for packets of flow k under the configuration where flow 1 uses path p_1 among its possible paths, flow 2 uses path p_2, and so forth.

The algorithm executed for flow k is probabilistic and maintains two vectors, both of size P_k (denoted P to ease notations).

The *probabilistic choice* vector $\mathbf{q} = (q_1 \dots q_P)$ gives the probability to choose each path p.

The *score vector* $\mathbf{Y} = (Y_1 \dots Y_P)$ maintains a (negative) score for each path, to be optimized, where Y_p depends on the average delay for packets of flow k on path p.

The main loop of the OPS algorithm (Algorithm 1) is as follows. Each time a local timer ticks, a path p is chosen according to the probability distribution $\mathbf{q}$, and packets are sent along p. The delay of packets over this path is measured. The score Y_p is updated according to a discounted sum and in turn, the probability vector is modified for the next path selection using a logit distribution. This repeats forever, or

until a stable path has been reached for all flows, *i.e.* $\mathbf{q}$ becomes a degenerate probability vector (all coordinates are equal to zero except one) for all flows.

Algorithm 1: Optimal Path Selection (OPS) Algorithm for flow k.

Initialize:
$n \leftarrow 0$; $\mathbf{q} \leftarrow (\frac{1}{P}, \ldots, \frac{1}{P})$; $\mathbf{Y} \leftarrow (0, 0, \ldots, 0)$;
repeat
  When local timer ticks for the nth time;
  $n \leftarrow n + 1$;
  select a new path p w.r.t. probabilities $\mathbf{q}$;
  Use path p and measure delay D;
  Update score of path p:
  $$Y_p \leftarrow \left(Y_p - \gamma_n(D + \tau Y_p)/q_p\right) \vee \beta_n;$$
  foreach *path* $s \in \mathcal{P}_k$ **do**
    update probability: $q_s \leftarrow \frac{\exp(Y_s)}{\sum_\ell \exp(Y_\ell)}$;
until *end of time*;

The OPS algorithm uses 3 parameters: $\tau > 0$ is a discounting factor over past scores. The bounding sequence β_n (in the algorithm, $\vee$ denotes the maximum operator) is such that $\beta_n \to -\infty$ when $n \to \infty$ and $|\beta_n| \leqslant C_1 n + C_2$ for some constants C_1 and C_2. The decreasing sequence of discretization steps γ_n is in L_2 ($\sum_n \gamma_n^2$ converges), but not in L_1 ($\sum_n \gamma_n$ diverges). Typically, $\gamma_n = 1/n^\alpha$ with $1/2 < \alpha \leqslant 1$ works.

C. Convergence Properties

Assumption 1. *The measurement $D(t)$ of the delay, done by flow k, over path p_k at physical time t is such that $D(t) = d_k(p_1(t) \ldots p_k(t) \ldots p_K(t)) + \xi(t)$, where the noise $\xi(t)$ is a non-biased random variable, conditionally to the past, $\mathcal{F}_t$: $\mathbb{E}(\xi(t)|\mathcal{F}_t) = 0$, and has a finite second moment: $\mathbb{E}(\xi(t)^2|\mathcal{F}_t) < \infty$.*

Recall that d_k is the average end-to-end delay experienced by flow k in a given configuration while $D(t)$ is an estimation based on actual measurements. This assumption simply means that estimates have no bias. This is a rather mild assumption, because this estimate does not need to be bounded, nor does it have to be independent of the current configuration of the traffic, for example a large load over one link may induce a noise with a larger variance on that link.

Assumption 2. *All the flows have the same arrival rate.*

This assumption is technical and deserves some comments. First, it is needed to make sure that the underlying game has a potential [14]. This is an essential ingredient to prove convergence. However, it is not realistic: In general, all flows do not have the same arrival rate. In that case, one can still use the OPS algorithm. It can be shown (not reported here) that if OPS converges, it finds a Nash equilibrium. However, convergence is not guaranteed in general. When the flows do not have the same rate, it is still possible to split flows into subflows, all with the same rate. In that case the subflows from the same flow may end up using different paths from source to destination.

Theorem 1 (Convergence to equilibrium)**.**
Under assumptions 1 and 2, for all $\varepsilon > 0$, there exists $\tau > 0$ such that under discounting factor τ, Algorithm 1 converges for all flows to an ε-optimal configuration, in the following sense:
For each flow k, the probability vector $\mathbf{q}$ converges almost surely to a near degenerate probability: q_p becomes smaller than ε for all $p \in \mathcal{P}_k$ except for one path, say p_k^*, for which it grows larger than $1 - \varepsilon$.
Furthermore, under configuration $(p_1^*, \ldots, p_K^*)$, no flow can unilaterally reduce its delay: for all k and $\forall p \in \mathcal{P}_k$, $d_k(p_1^*, \ldots, p, \ldots, p_K^*) \geqslant d_k(p_1^*, \ldots, p_k^*, \ldots, p_K^*)$.

The proof of Theorem 1, given in the Appendix, also shows the following additional properties of the OPS Algorithm.

The algorithm is incrementally deployable. Not all flows in the network need to use the OPS algorithm. If only a subset of the flows use the OPS algorithm, while all the other flows are forwarded using static routing tables, the convergence property still holds. The statically routed flows are seen as part of the (random) environment over which optimization is done.

OPS is only based on online local information. The only information used by the flows is the current delay on their current path. They do not know the topology of the network, the capacity of the network, the number of concurrent flows or the paths taken by the other flows. The discovery of the best path for each flow is done "in the dark", without any exchange of information or any kind of collaboration between the flows or with the network.

Asynchronous updates. The algorithm is truly asynchronous: convergence will occur even if the different flows update their choices using different timers, that can have arbitrary initial offsets as well as arbitrary different speeds. Of course the speed of convergence may be slow if flows have very different update frequencies.

Robustness to errors. As explicitly specified in Assumption 1, convergence to the equilibrium is robust to measurement errors. Actually, this is very useful in our context, not only because measurements do not have an infinite precision but mainly because the measurement of the delay over one path is done by monitoring one or several packets (see Section III-B for a detailed description on how this can be done) and by computing the empirical mean. This empirical mean equals the average delay up to some additive random term. This term satisfies Assumption 1 as soon as the system is stationary.

Robustness to outdated measurements. The estimation of the delay D on path p is based on monitoring the delay of several packets taking this path. When the score Y_p is computed and a new path is selected, these measurements are outdated because other flows may have already changed their

paths. We can show that this phenomenon does not jeopardize the convergence properties of the algorithm.

Finally, here are some complementary comments.

a) Relaxing convergence: The guarantee of almost sure convergence requires that the step-size sequence γ_n vanishes to 0. One can relax this vanishing condition and show that for constant step sizes ($\gamma_n = \gamma, \forall n \in \mathbb{N}$), convergence occurs in probability instead of almost surely. Namely, the distribution of $\mathbf{q}$ concentrates to near degenerate probability distributions when γ and τ go to 0.

b) Speed of convergence: The proof of the theorem is based on the construction of a stochastic approximation of a differential equation and does not provide any information on the speed of convergence of the algorithm to a Nash equilibrium. The speed convergence of similar algorithms has been proved to be of order $1/n$ [15] in a centralized context (a single player) and with strictly convex objective functions. Neither conditions are true here (many players with arbitrary delay functions). To our knowledge, no theoretical result exist today to bound the speed of convergence for the type of algorithms used here, so that it must be evaluated experimentally.

On a practical point of view, several factors can affect the convergence of the OPS algorithm.

- Size of the network: The algorithm being based on the measurement of end-to-end delays, the size of the network should not play a decisive role in the convergence speed. The effects of a larger network are indirect: it increases the number of possible routes for each flow, and potentially increases the amount of outdated information.
- Number of players: One of the main advantages of the OPS algorithm with respect to classical ones where players play one at a time, is that simultaneous play speeds up convergence, especially with a large number of players. Several numerical simulations of OPS done in a different context [13] (not reported here) suggest that the speed of convergence does not depend crucially on the number of players.
- Number ot routes per player: In contrast with the number of players, the convergence time should increase drastically when the number of choices per flow increases: Each flow needs to experiment each route a sufficient number of times before it can assess its performance reliably.
- Precision of the measurements: It shoud be clear that when the measurements of the end-to-end delays suffer from a large variance, then convergence gets slow. Actually, we believe this is the most important parameter that influences the convergence time. This belief is reinforced by the experiments reported in Figure 5.

c) Price of Anarchy: In general, a Nash equilibrium (NE) does not provide any guarantee on its global performance. In the worst case, the sum of the delays of all flows under a Nash equilibrium can be arbitrarily far from an optimal configuration.

A first argument in favor of NE is that social optima that are not NE are unstable configurations, because some flows can gain by changing their paths, and will certainly do so if they are not constrained by some kind of central controller.

Moreover, in many cases, NE exhibit good global performance. This has been deeply investigated in the literature: we simply provide a quick review here. In general, the *price of anarchy* (ratio between the social cost of the worse Nash equilibrium over the cost of the social optimum) is bounded by $\Theta(\log n)$ [16] where n is the number of players.

Furthermore, if the delays of all links are (λ, μ)-smooth (see the definition in [16]), then the price of anarchy is bounded by a constant, namely $\frac{\lambda}{1-\mu}$. As a special case, this bound becomes $5/2$ when the delay on each link of the network is affine w.r.t. its load (see [16]).

Finally, NE become optimal over arbitrary networks when the load goes to one or when the size of the network grows to infinity [17].

III. Implementation as a SDN controller

This section examines all the features of our model and of Algorithm 1, and explains how the latter can be implemented as a SDN controller.

A. Running our algorithm in gateways

Algorithm 1 assumes that flows, or equivalently end-hosts, can choose the full path to their destination (line 6 of the algorithm). This paradigm is known as *source routing*, but is not widely deployed in the public Internet: source routing was deprecated in IP because of security issues [RFC 5095]. Thus, we first make assumptions on the structure of the network, and we then adapt Algorithm 1 so that routing decisions are taken by routers on behalf of end-hosts.

First, we assume that the network can be decomposed into a *core* network and several *edge* networks. Each edge network connects to the rest of the network through a single router, which we call a *gateway*. Gateways may connect directly to each other, or through core routers. Figure 2 shows a simple example, with four gateways (G_1 to G_4) and a single core router R. In this example, each edge network only contains one host beside its gateway.

Then, to ease implementation, we assume our OPS algorithm is only run by gateways to select a path among those offered by the core network. This way, core routers do not need to keep state for each source-destination flow. Core routers simply use their regular routing protocols, for instance BGP or OSPF. They can also use ECMP with a hashing mechanism on the packet source and destination, so that a given source-destination flow in the core network is forwarded along a unique path. This is necessary to obtain consistent delay measurements during the lifetime of a flow.

Limiting our algorithm to only run in gateways means that a flow cannot explore all possible paths to its destination: the set of usable paths $\mathcal{P}_k$ defined in Section II for flow k is restricted to the next-hop routers of the gateway. Several examples [18], [19] show that substantial gains can still be obtained with only two or three paths.

B. Exploiting one-way delay

Algorithm 1 uses a generic notion of "delay" as objective function (line 7 of the algorithm). We focus on end-to-end *one-way delay* instead of the more classical RTT. There are several reasons to this choice: 1) For latency-critical applications such as Voice-over-IP, the relevant metric is often the transmission delay on the forward path. 2) A router has no control over the reverse path since routing decisions only affect the *forward* path of a flow towards its destination. For instance, the LEDBAT congestion control algorithm [20] exploits one-way delay for the same reason: it allows congestion detection and bufferbloat avoidance on the forward path, while ignoring the effect of the reverse path.

To passively determine the one-way delay of forwarded packets, we use *timestamps* that some transport protocol include in their headers: we focus on TCP with the Timestamp Option and μTP, an implementation of LEDBAT [20] over UDP. To estimate one-way delay from these timestamps, we use the same method as [21] to extract one-way delay samples from TCP and μTP/UDP packets on the reverse path, and then normalize the measurements by expressing them in milliseconds.

In our case, the gateway is the "observation point" at which the delay is observed. The gateway measures the one-way delay from A to B, where A is a host in the local edge network and B is a remote host. For every packet i from A to B, the gateway can directly measure the *observed one-way delay* $\delta_{\text{observed}}(i)$, which can then be theoretically decomposed as:

$$\delta_{\text{observed}}(i) = \delta_{\text{propag.}}(p_i) + \delta_{\text{queuing}}(i) + \text{off}_{A,B}$$

where p_i is the path taken by packet i in the network, $\delta_{\text{propag.}}(p_i)$ is the one-way propagation delay from A to B using path p_i, $\delta_{\text{queuing}}(i)$ is the total queuing delay on the path from A to B for packet i, and $\text{off}_{A,B}$ is the clock offset of B relative to A.

To eliminate the clock offset and get positive estimates of the delays (required for proper convergence of the algorithm), we compute the minimum of the observed delay measurements, similarly to LEDBAT. In our case, this minimum is taken over all paths selected by the algorithm. Given a measurement of the observed delay $\delta_{\text{observed}}(i)$ on the ith packet using path p_i, the *relative delay* we consider for our algorithm is the following:

$$\Delta(i) := \delta_{\text{observed}}(i) - \min_{0 \leqslant l \leqslant i} \delta_{\text{observed}}(l)$$

$$\Delta(i) = \delta_{\text{observed}}(i) - \min_{0 \leqslant l \leqslant i} \left[\delta_{\text{propag.}}(p_l) + \delta_{\text{queuing}}(l) + \text{off}_{A,B}\right] \tag{1}$$

$$= \delta_{\text{observed}}(i) - \left[\text{off}_{A,B} + \min_{\text{path } p} \delta_{\text{propag.}}(p)\right] \tag{2}$$

$$= \delta_{\text{propag.}}(p_i) + \delta_{\text{queuing}}(i) - \min_{\text{path } p} \delta_{\text{propag.}}(p). \tag{3}$$

Going from (1) to (2) assumes that i is large enough to achieve a minimum queuing delay of zero, and to ensure all paths have been visited. In practice, our algorithm starts with a uniform probability to select each path, so most paths are expected to be probed before the algorithm starts to converge.

This way, we eliminate the clock offset and obtain a positive estimate of the one-way-delay on the current path p_i, relative to the path with the lowest propagation delay. It can be shown that this additive shift does not affect the behavior of the algorithm, given that $\Delta(i)$ remains non-negative.

C. Update policy

Algorithm 1 (line 4) uses a timer for its updates, denoted n. This becomes a periodic update in the implementation: for each flow forwarded by a gateway, the gateway takes a routing decision every T packets of this flow. Among these T packets, the gateway only uses the one-way delay of the last S of them and computes their empirical mean. More precisely, $\Delta(i)$ being the *relative delay* of packet i defined above, the delay D used in line 7 of Algorithm 1 is computed as the average of $\Delta(i)$ for the last S packets:

$$D := \big(\Delta(nT - S + 1) + \cdots + \Delta(nT)\big)/S.$$

This choice has several advantages:

1) Large flows, with a heavy influence on the network, are updated more frequently.
2) This controls the random error ξ defined in Assumption 1: a large T ensures that the expectation of ξ is close to 0 while a large S reduces its variance. Convergence to the stationary state over the new path is exponentially fast when the network is stable (see for example [22]). Thus, if we only measure the delay for the last σ packets among the ν packets, and if σ is sufficiently smaller than ν, then the expected delay for these packets is very close to the stationary average delay. Furthermore, this empirical mean has no bias and satisfies Assumption 1. Finally, the empirical mean over the last σ packets is close to the average delay, since the standard deviation of the error decreases with the square root of the sample size σ (central limit theorem). Note that formally, the central limit theorem cannot be applied here, because our samples on the delays are not independent. However, it still provides a useful heuristic estimation of the gap between empirical mean and average delay.

D. SDN-based implementation

Given the discussion above on implementation issues, we adapted Algorithm 1 so that it is more practical for real networks. Our adapted algorithm is more formally described in our research report [12], and the code itself is available online[1].

Our proof-of-concept implementation takes the form of an Openflow controller, using the Ryu [23] library. A gateway is made of an Openflow switch and a dedicated controller, but we use the term "gateway" to refer to the Openflow switch only.

[1] `https://gforge.inria.fr/projects/derouted`

The forwarding table of a gateway is programmed and constantly updated by its controller. Furthermore, a gateway sends a copy of packet headers back to its controller, for delay computation. The controller is configured beforehand with a static multipath routing table. That is, for each IP prefix, the routing table lists all possible next-hop routers that can be used to reach the destination.

When a gateway receives the first packet of a new flow, it forwards it to the controller, to receive instructions for subsequent packets of the same flow. The controller takes an initial routing decision for this flow, according to its routing table. This decision is a choice of a next-hop router through which packets will be forwarded. A corresponding forwarding rule is then installed into the gateway.

To measure the performance of each decision, the controller also installs a rule to receive a copy of all packet headers. Upon receiving a header, the controller extracts timestamps, computes the end-to-end one-way delay of the flow, and updates the score of the current choice. For each active flow independently, the controller periodically decides to select a new next-hop router, based on the scores. Each time the controller changes its decision, new forwarding rules are installed in the gateway.

The main reason to use a SDN framework is ease of implementation. Indeed, Openflow abstracts away the communication with forwarding elements, allowing to easily install forwarding rules and receive packet headers for performance measurements. For our experiments, we used Openvswitch on Linux, but any Openflow-compatible switch could have been used instead.

An important architectural decision was to exploit the decentralized nature of our algorithm, by ignoring the centralized management features of Openflow. Here, each gateway is controlled by a separate Openflow controller. Thus, we retain the scalability and resilience properties of decentralized routing.

Another key point is that we use Openflow switches as pure layer-3 devices. Openflow 1.3 is perfectly capable of programming a switch to act as a layer-3 router, by decreasing the TTL and modifying the MAC addresses of packets. Working at layer 2 would not make much sense, because we fundamentally solve a routing problem. Besides, it is difficult to have control on flooding when working at layer 2 in complex topologies with redundant paths.

IV. Experimental Evaluation

We next evaluate our OPS algorithm in a simple network, to make sure it discovers optimal paths in practice. We are also interested in the time it takes to reach a solution.

The experiments were performed with emulation on Linux systems. This means that we were able to run our SDN implementation unmodified: we thus expect our results to be quite close to reality. Unfortunately, it also means that we were unable to perform a high number of experiments or use large-scale topologies. Indeed, setting up and then running each experiment in a reproducible manner requires significant time and efforts.

We used Mininet [24] to emulate network topologies, running on a single server. Although realistic in a functional sense, Mininet is known to exhibit unrealistic performance characteristics. To prevent this effect, we limit the capacity of the links to very low values using netem (in the order of 10 Mbit/s). Furthermore, we run our Mininet network on fast and modern server hardware, which should alleviate performance concerns given the low throughput involved in the experiments.

A. Experimental setup

To evaluate our algorithm, we consider a network satisfying the conditions laid out in Section III-A. The resulting network is shown in Figure 2.

For this evaluation, all flows have a constant throughput equal to λ, and have an infinite duration. It can be seen as a model of a particular application usage, for instance a network used to transport multiple real-time video streams.

The central router R simply forwards packets using a static routing table, designed to minimize the number of hops to the destination.

Additionally, each gateway uses static routes for packets coming from other gateways, to avoid routing loops. The next-hop router of these static routes is the destination gateway if it is a neighbor, and R otherwise.

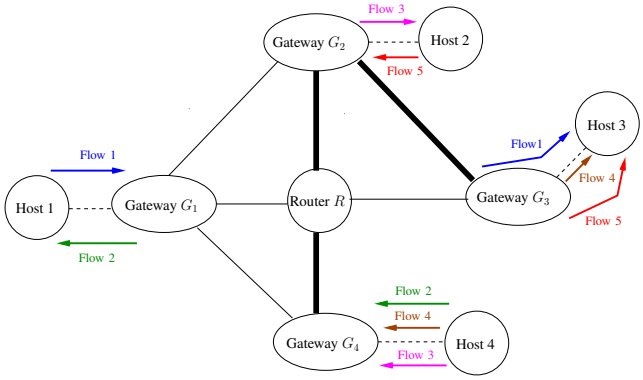


Fig. 2: Network used in the following experiments. Thin lines represent links with capacity of 4 Mbit/s, while thick lines represent links with a capacity of 8 Mbit/s. All links have a transmission delay equal to 5 ms. Dashed lines between hosts and their gateway are links with unrestricted capacity. Five flows using the network are represented. Each flow consists in UDP packets with a constant throughput λ, with λ varying from 2000 to 3900 Kbit/s in the experiments.

The capacity of each link is detailed in the caption of the Figure. To ease explanations, we introduce the *load* ρ as $\rho = \frac{\lambda}{4000 \text{ Kbit/s}}$. A load of 1 means that a single flow would occupy the full capacity of a 4 Mbit/s link.

As long as $\rho < 1$, there exists at least one stable configuration of the network, *i.e.* a choice of path for each flow

that satisfies capacity constraints on all links. Additionally, for $\frac{2}{3} < \rho < 1$, there are only two possible stable configurations, described in Figure 3.

Flow	Equilibrium 1	Equilibrium 2
Flow 1	$G_1 \to G_2 \to G_3$	$G_1 \to G_2 \to G_3$
Flow 2	$G_4 \to G_1$	$G_4 \to R \to G_1$
Flow 3	$G_4 \to R \to G_2$	$G_4 \to R \to G_2$
Flow 4	$G_4 \to R \to G_3$	$G_4 \to G_1 \to R \to G_3$
Flow 5	$G_2 \to G_3$	$G_2 \to G_3$

Fig. 3: Stable configurations for the network of Figure 2, when the load $\rho = \frac{\lambda}{4000 \text{ Kbit/s}}$ satisfies $\frac{2}{3} < \rho < 1$.

As described above, we use Mininet to emulate the network topology described in Figure 2. Mininet's topology API allows us to describe our topology at an abstract level, by creating nodes and links with specific properties. The topology is then instantiated in a single physical machine running Linux.

In this topology, our gateways are implemented as Openflow switches managed by `openvswitch`, each of which is connected to a dedicated controller. Router R is simply a host with multiple interfaces, and a static routing table used to forward packets between interfaces. To emulate links with a limited capacity and a specified delay, we use the `TCIntf` class provided by Mininet. Internally, it uses the `htb` queuing discipline provided by Linux to limit the capacity, and `netem` to emulate a transmission delay.

Since we have four gateways, we run four instances of our Ryu-based Openflow controller, each controlling a different emulated gateway. Once Mininet and the controllers are setup, we then use `udpmt` (part of the `ipmt` toolbox [25]) to generate UDP packets for each flow. We modified `udpmt` so that it sends packets with a μTP header, which includes a timestamp of the date of emission. The destination host for each flow runs `udptarget`, which we modified to behave similarly to a μTP implementation: it replies back with small UDP packets containing μTP timestamps.

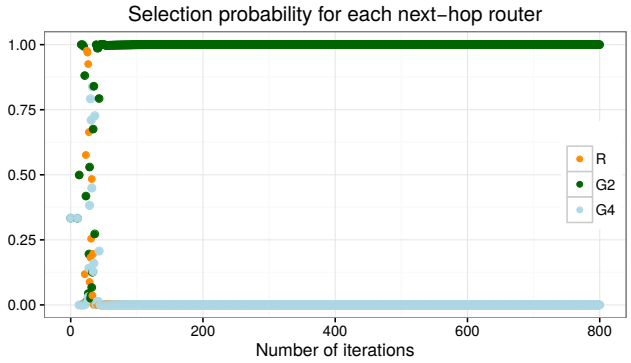
This setup has been run a large number of times, to reduce variability. Each experiment lasts for 2400 seconds, because we found it was sufficient to allow most executions to converge. To parallelize the execution, we used Grid'5000 [26] as an IaaS platform. We reserved several identical physical machines from a cluster, deployed a Debian Jessie image with all necessary software, and ran the same experiments independently on each machine. The machines are Bullx Blade B500 servers, with a 8-core Intel Xeon E5520 CPU and 24 GiB of RAM. Depending on the experiments, we used between 5 and 45 machines in parallel.

B. Discussion of the results

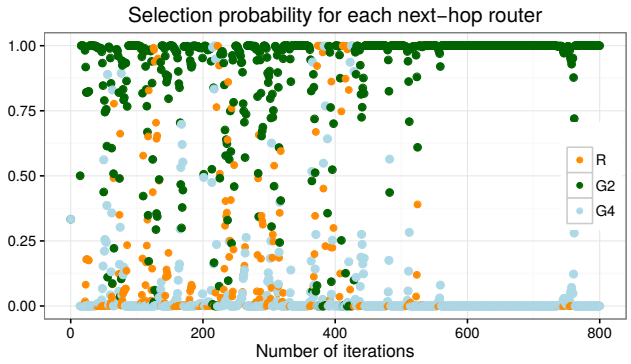
The experiment was carried under several values of load ρ. In the experiments, the load varies from 0.75 to 0.975, to exercise the algorithm under moderate and heavy loads.

Figure 4 shows the evolution of Flow 1 under different loads. Gateway G_1 has three possible next-hop routers: G_2, R, or G_4. We display the probability, over time, that G_1 selects each of the three next-hop routers. The other flows are also being forwarded concurrently, but this is not displayed here.

G_2 gets selected most of the time, after a transient period. This choice is consistent with both optimal configurations for the network. We also note that, when the load increases, the transient period becomes longer.



(a) Load equal to 0.75



(b) Load equal to 0.925

Fig. 4: Probability of selecting each next-hop router over time, for Flow 1 (see Figure 2). Each figure corresponds to an increasing value of load for all flows. Here, $T = 500$ packets and $S = 5$ packets.

Looking at Figure 4b, the convergence of Flow 1 is not obvious when the load is high. In fact, Flow 1 converges to an "almost pure choice" as predicted by Theorem 1. This means that the flow will spend most of its time on the optimal choice, but can explore other choices from time to time. This effect is more visible when the load is high: in Figure 4b, G_2 is selected most of the time, but the flow also explores G_4 and R from time to time.

Recall from Section III-C that a gateway periodically updates the path of its flows (this happens every $T = 500$ forwarded packets in our experiments). Taking this into account, we have designed the following convergence criterion for a flow: *A flow has converged to path p at time t if the average probability of choosing p for the last W updates was at least* Θ. Here, W is the size of a moving window counted as a

number of updates of our algorithm. The *global convergence criterion* for a network with multiple flows is satisfied when *all* flows satisfy the convergence criterion at time t, for the same parameters W and Θ.

Using this convergence criterion over the previous experiments with $W = 50$ and $\Theta = 80\%$, we have computed the global convergence time of the algorithm under different loads. Note that the convergence time cannot be lower than $W = 50$ iterations. The results are depicted in Figure 5, where the convergence time is expressed as a number of iterations of our algorithm. For a moderate load, the global convergence time is close to the minimum of $W = 50$ iterations. When the load approaches 1, we expect that the global convergence time goes to infinity, because the network cannot satisfy all demands and becomes unstable. Note that we intentionally exercise our algorithm in unfavorable conditions. For a low or moderate load, which is the usual case for over-provisioned operator networks, the global convergence time is very close to the minimal convergence time.

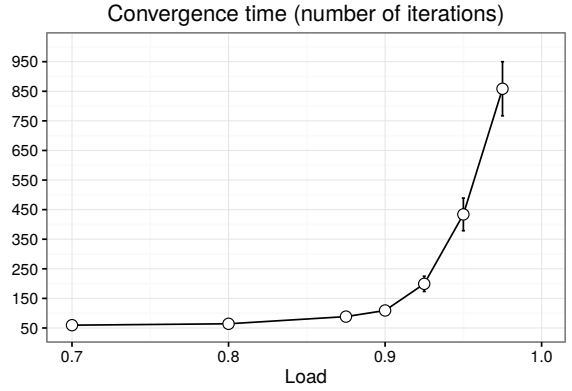


Fig. 5: Average global convergence time of the algorithm as a function of the load, with 95 % confidence intervals.

C. Quality of the final configurations

In the network of Figure 2, there exist two stable configurations (in other words Nash equilibria) where no flow can lower its delay by changing its path. These two configurations are given in Figure 3.

Only Flows 2 and 4 differ between these two configurations. In the second configuration, both flows choose a longer path. Thus, the first configuration is better than the second one, because the average delay of any flow is either smaller or unchanged. However, both configurations successfully avoid congestion. This explains why the algorithm may end up in the second configuration.

The following table shows the percentage of experiments that converged to equilibrium 1, equilibrium 2, any non-stable configuration, or did not converge. As the load increases, it can be observed that equilibrium 1 (the best configuration) is more likely to be selected by our algorithm, which matches intuition.

Load	0.7	0.8	0.875	0.9	0.925	0.95	0.975
Eq. 1	54.1	56.5	65.9	68.2	76.5	83.5	58.3
Eq. 2	44.7	43.5	31.8	31.8	21.2	15.3	17.9
Other	1.2	0.0	2.4	0.0	2.4	1.2	13.1
No cv.	0.0	0.0	0.0	0.0	0.0	0.0	10.7

For each value of network load, 85 independent experiments were conducted. The results in the table use our convergence criterion with $\Theta = 0.8$ and $W = 50$. The executions reported as non-convergent had failed to converge after the end of the experiment (2400 seconds).

The quality of the result actually depends on the parameters Θ and W of the stopping criterion. If one chooses the threshold Θ closer to one, then the quality improves, in the sense that the number of times that the OPS algorithm stops under an unstable configuration drops. However this comes at a cost: the execution times increases as well as the number of executions that do not finish before the time horizon. An empirical search (not reported here) with varying values of Θ and W shows that a good compromise in our experiment is obtained by choosing $\Theta = 0.8$ and $W = 50$.

V. Conclusion and future work

We have described our adaptive routing algorithm for packet-switched networks, first as a theoretical algorithm, and then as a practical implementation that can be used as a Openflow controller. In an idealized network model, our algorithm converges to an ε-Nash Equilibrium, as defined in Theorem 1. Using our prototype SDN implementation in an emulated network, we have then shown that it converges fast under light to moderate network load, and indeed reaches a Nash Equilibrium (i.e. a stable configuration where no flow can improve its delay). Under very heavy network load, convergence is slower, which is not surprising given the variability of delays when the network becomes saturated. However, even in these extreme conditions, our implementation still converges to an ε-Nash Equilibrium in most cases.

Appendix

This appendix is devoted to a sketch of the proof of Theorem 1. The proof is mainly based on the fact that the OPS algorithm is a variant of an algorithm presented in [13], that computes Nash equilibria in general potential games. In addition to this main ingredient, it also uses results from [27] to guarantee convergence even when the scores are not bounded, and results from [28] to guarantee convergence to pure NE.

First, let us show that our routing problem can be seen as a game. The flows are the players, their strategies are the choices of paths, and the payoffs (or the costs, here) are the expected delays on the chosen paths. Under this framework, our problem fits in a well-known class of games, namely *atomic routing games*. In [14], it was shown that atomic routing games where each player (flow) has the same rate are *potential games*, *i.e.* there exists a *potential function* ϕ for such games, namely,

$$\phi(\mathbf{p}) = \sum_{e} \sum_{i=1}^{N_e(\mathbf{p})} \delta_e(i),$$

where $\delta_e(i)$ is the delay on link e when i flows use it, and $N_e(\mathbf{p})$ is the number of flows using link e under configuration $\mathbf{p}$.

Assumption 2 implies that our problem is indeed a potential game as described above.

Using this construction, our algorithm is similar to Algorithm 1b in [13] with the addition of bounding terms β_n.

The L2-L1 condition on γ_n is the same as assumption A1 in [13]. Our Assumption 1 implies Assumption A2 in [13]. As for assumption A3 in [13], it is replaced here by the explicit bounds β_n. The bounded approximation theorem (Theorem 2) in [27] allows us to state that the sequence Y_n is an *asymptotic pseudo-trajectory* (APT) in the sense of [29] as soon as the clock ticks for the flows all have finite rates. This implies that for each flow the scores of paths, Y_p, approach the solution of the following differential system (4)-(5), even if the estimation D is based on outdated measurements of packet delays

$$\frac{dy_p}{dt} = d_p(\mathbf{q}) - \tau y_p, \quad \forall p \in \mathcal{P} \tag{4}$$

$$q_p = \frac{\exp(y_p)}{\sum_\ell \exp(y_\ell)} \quad \forall p \in \mathcal{P}, \tag{5}$$

as long as this solution is locally stable (in the dynamical system sense).

In [13], is is shown that the rest points of this differential system are locally stable when the game is a potential game and are $\varepsilon(\tau)$ approximations of Nash equilibria of the game, where the gap $\varepsilon(\tau)$ vanishes as $\tau \to 0$. This means that Theorem 2 in [27] can be used here.

Finally, the existence of a potential function further implies that the only locally stable Nash equilibria are pure (see [28], [30]).

Putting everything together implies that our algorithm will converge arbitrarily close to a pure Nash equilibrium. In our case the pure Nash equilibria are configurations that contain a path p_k for each flow k such that

$$d_k(p_1, \ldots, p_k, \ldots, p_K) \leqslant d_k(p_1, \ldots, p', \ldots, p_K),$$

for all p' in $\mathcal{P}_k$.

References

[1] T. C. Hu, "Multi-commodity network flows," *Operations Research*, vol. 11, no. 3, pp. 344–360, 1963.

[2] B. Awerbuch and R. D. Kleinberg, "Adaptive routing with end-to-end feedback: Distributed learning and geometric approaches," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pp. 45–53, ACM, 2004.

[3] S. Bubeck and N. Cesa-Bianchi, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Machine Learning*, vol. 5, no. 1, pp. 1–122, 2012.

[4] W. Krichene, *et al.*, "On the convergence of no-regret learning in selfish routing," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 163–171, 2014.

[5] X. Wu and X. Yang, "Dard: Distributed adaptive routing for datacenter networks," in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pp. 32–41, IEEE, 2012.

[6] W. Cui and C. Qian, "Difs: Distributed flow scheduling for adaptive routing in hierarchical data center networks," in *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pp. 53–64, ACM, 2014.

[7] S. Barre, *et al.*, "MultiPath TCP: From theory to practice," in *NETWORKING 2011* (J. Domingo-Pascual, *et al.*, eds.), vol. 6640 of *Lecture Notes in Computer Science*, pp. 444–457, Springer Berlin Heidelberg, 2011.

[8] B. Veal, *et al.*, "New methods for passive estimation of TCP round-trip times," in *Passive and Active Network Measurement*, pp. 121–134, Springer, 2005.

[9] A. Khanna and J. Zinky, "The revised arpanet routing metric," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 45–56, 1989.

[10] B. Fortz and M. Thorup, "Optimizing ospf/is–is weights in a changing world," *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, vol. 20, no. 4, 2002.

[11] A. Kvalbein, *et al.*, "Multipath load-adaptive routing: Putting the emphasis on robustness and simplicity," in *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*, pp. 203–212, IEEE, 2009.

[12] B. Jonglez and B. Gaujal, "Distributed Adaptive Routing in Communication Networks," Research Report RR-8959, Inria ; Univ. Grenoble Alpes, Oct. 2016.

[13] P. Coucheney, *et al.*, "Penalty-Regulated Dynamics and Robust Learning Procedures in Games," *Mathematics of Operations Research*, vol. 40, no. 3, pp. 611–633, 2015.

[14] A. Orda, *et al.*, "Competitive routing in multiuser communication networks," *IEEE/ACM Trans. on Networking*, vol. 1, no. 5, pp. 510–521, 1993.

[15] A. S. Nemirovski and D. B. Yudin, *Problem Complexity and Method Efficiency in Optimization*. New York, NY: Wiley, 1983.

[16] G. Christodoulou and E. Koutsoupias, "The price of anarchy of finite congestion games.," in *37th Annual ACM Symposium on Theory of Computing (STOC)*, 2005.

[17] R. Colini-Baldeschi, *et al.*, "On the price of anarchy of highly congested nonatomic network games," in *9th International Symposium of Algorithmic Game Theory (SAGT)* (Springer, ed.), no. 9928 in LNCS, pp. 117–128, 2016.

[18] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 1094–1104, Oct. 2001.

[19] P. Coucheney, *et al.*, "Self-optimizing routing in manets with multiclass flows," in *Personal Indoor and Mobile Radio Communications (PIMRC), 2010 IEEE 21st International Symposium on*, pp. 2751–2756, IEEE, 2010.

[20] S. Shalunov, *et al.*, "Low Extra Delay Background Transport (LEDBAT)." RFC 6817 (Experimental), Dec. 2012.

[21] C. Chirichella, *et al.*, "Passive bufferbloat measurement exploiting transport layer information," in *Global Communications Conference (GLOBECOM), 2013 IEEE*, pp. 2963–2968, IEEE, 2013.

[22] D. Gamarnik and S. Meyn, "On exponential ergodicity of multiclass queueing networks," *Queueing Systems*, vol. 65, no. 2, pp. 109–133, 2010.

[23] "Ryu: a component-based software defined networking framework." http://osrg.github.io/ryu/.

[24] B. Lantz, *et al.*, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19, ACM, 2010.

[25] G. B. Sabbatel and M. Heusse, "ipmt: Internet protocols measurement tools." https://forge.imag.fr/projects/ipmt/.

[26] D. Balouek, *et al.*, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science* (I. Ivanov, *et al.*, eds.), vol. 367 of *Communications in Computer and Information Science*, pp. 3–20, Springer International Publishing, 2013.

[27] S. Andradóttir, "A stochastic approximation algorithm with varying bounds," *Operations Research*, vol. 43, no. 6, pp. 1037–1048, 1995.

[28] W. H. Sandholm, *Population Games and Evolutionary Dynamics*. MIT Press, 2010.

[29] M. Benaïm, "Dynamics of stochastic approximations," in *Séminaire de Probabilités*, vol. 1709 of *Lectures Notes in Mathematics*, pp. 1–68, 1999.

[30] D. Monderer and L. Shapley, "Potential games," *Games and economic behavior, Elsevier*, vol. 14, no. 1, pp. 124–143, 1996.