Optimal Local Buffer Management for Information Gathering with Adversarial Traffic*

Stefan Dobrev†
Inst. of Mathematics
Slovak Academy of Sciences
Bratislava, Slovakia
Stefan.Dobrev@savba.sk

Manuel Lafond‡
School of Eng. and Comp, Sci.
Université d'Ottawa
Ottawa, Canada
lafonman@iro.umontreal.ca

Lata Narayanan§
Dept. Comp. Sci. & Soft.Eng.
Concordia University
Montréal, Canada
lata@cs.concordia.ca

Jaroslav Opatrny
Dept. Comp. Sci. & Soft.Eng.
Concordia University
Montréal, Canada
opatrny@cs.concordia.ca

ABSTRACT

We consider a problem of routing on directed paths and trees to a single destination, with rate-limited, adversarial traffic. In particular, we focus on local buffer management algorithms that ensure no packet loss, while minimizing the size of the required buffers.

While a centralized algorithm for the problem that uses constant-sized buffers has been recently shown [21], there is no known *local* algorithm that achieves a sub-linear buffer size. In this paper we show tight bounds for the maximum buffer size needed by ℓ-local algorithms for information gathering on directed paths and trees, where an algorithm is called ℓ-local if the decision made by each node v depends only on the sizes of the buffers at most ℓ hops away from v.

We show three main results:

- a lower bound of $\Omega(c \log n/\ell)$ for all ℓ-local algorithms on both directed and undirected paths, where c is an upper bound on the link capacity and injection rate.
- a surprisingly simple 1-local algorithm for directed paths that uses buffers of size $O(\log n)$, when $c = 1$.
- a natural 2-local extension of this algorithm to directed trees, for $c = 1$, with the same asymptotic bound.

Our $\Omega(\log n)$ lower bound is significantly lower than the $\Omega(n)$ lower bound for greedy algorithms, and perhaps surprisingly, there is a matching upper bound. The algorithm that achieves it can be summarized in two lines: If the size of your buffer is odd, forward a message if your successor's buffer size is equal or lower. If your buffer size is even, forward a message only if your successor's buffer size is strictly lower. For trees, a simple arbitration between siblings is added.

CCS CONCEPTS

• **Theory of computation** → *Network flows*;

KEYWORDS

Buffers, Buffer size, Routing, Trees, Directed paths, Information gathering, Local algorithms, Adversarial traffic

1 INTRODUCTION

Buffer or queue management in packet-switched networks has been an extensive subject of study for decades. Early theoretical work in the area studied *static* routing problems; the source-destination pairs corresponding to a finite set of packets is given as input to the network, and the goal is to route packets from their sources to their respective destinations, while minimizing the worst-case arrival time as well as the maximum size of buffer needed. In the case when multiple routes use the same link, a node may need to store incoming packets in a buffer, and to use a *buffer management* or *scheduling* policy that dictates which packet, if any, should be forwarded along each output port in each step. Well-known examples of scheduling policies include First-In-First-Out (FIFO), Last-in-First-Out (LIFO), Furthest-to-Go (FTG), Nearest-to-Go (NTG), etc. The policy used for buffer management has an impact on many crucial quality-of-service parameters for networks.

More recently, buffer management has been studied in the context of *dynamic* routing, where packets are continuously injected into the network. In a seminal paper, Borodin *et al* [11] introduced an *adversarial* model for traffic to analyze the *worst-case performance* of a scheduling strategy for dynamic routing. In this model, time proceeds in discrete steps. Given a network, in every step, an adversary injects packets at a certain set of nodes, and specifies, for each packet, a path to a destination, where it is *consumed*. The scheduling policy

*
†Supported by VEGA grant
‡Supported in part by NSERC grant.
§Supported in part by NSERC grant.

SPAA '17, July 24-26, 2017, Washington, DC, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4593-4/17/07... $15.00
https://doi.org/10.1145/3087556.3087577

now chooses at most one packet to forward over each link of the network. Clearly, the network would be overwhelmed if the adversary generates more packets than can be sustained by the bandwidth in the network. Therefore, the adversary is assumed to be *rate-constrained*.

A key question is whether a given scheduling policy is *stable* for a given network, i.e., whether the sizes of the buffers remain bounded. One class of scheduling strategies that has been extensively studied is the so-called *greedy* or *work-conserving* policies, wherein a packet is always forwarded along an edge e if there are packets waiting to use e. It has been shown that there are work-conserving policies that are stable; however, the worst case buffer size can be polynomial in the size of the network. Even for a path, the worst-case buffer size for the greedy algorithm is $\Omega(n)$ [23].

In this paper, we study a class of routing problems called *information gathering* or *convergecast*, where the network has a special node called the *sink node*, and all packets generated in the network are destined for the sink. Such a communication pattern has been widely studied, particularly in the case of sensor networks, where sensor nodes collect data and forward it to a sink node for processing. We are interested in ℓ-*local* scheduling policies: every node must make its decision based on the contents of its own buffer, and knowledge of the buffer sizes of nodes in its ℓ-neighborhood. The goal of our work is to find upper and lower bounds on the buffer size required to achieve convergecast on trees with *no packet loss*, while using a *local* scheduling policy.

1.1 Related work

Adversarial queuing theory was introduced in [11] as a new approach to the study of queueing networks in general, and in particular, to the performance of scheduling algorithms in the context of dynamic packet routing in a network. The authors proposed a fixed-rate adversary to generate the input consisting of the nodes where new packets are injected into the network, together with specific paths to their respective destinations. The main question considered in [11] was the *stability* of a queuing discipline for a particular network, *viz.* given a network G and a scheduling policy $\mathcal{S}$, is there a constant M (which can depend on the size of the network but is independent of the length of the input stream) so that for any input stream, the size of all buffers in the network remain bounded by M? Related questions of interest that were posed were the existence of universally stable policies, i.e. stable for all networks, and universally stable networks, i.e. stable for all policies of a given class. It was shown in [11] that every greedy queueing discipline is stable for rate 1 adversaries on any DAG, and that Furthest-to-Go is stable for rate 1 adversaries in a uni-directional ring. Andrews et al [5] extended the result by showing that every greedy queueing discipline is stable for rate 1 adversaries in a unidirectional ring. They also showed that certain scheduling policies, such as Farthest-to-Go (FTG), Nearest-to-Source (NTS), Longest-in-System (LIS), and Shortest-in-System (SIS) are *universally stable*, while common policies such as FIFO, LIFO, and NTG, and Farthest-from-source (FTS) are not. However, the aforementioned policies were shown to require queues and delays of size exponential in the size of the network in the worst case. Finally, they give a local distributed and randomized scheduling policy that uses polynomial size buffers in the worst case. For further studies of this problem see for example, [3, 4, 6, 8, 10, 12, 19, 20].

Aeillo et al [2] proposed the related *Competitive Network Throughput* model in which the buffer size at every node is fixed to a constant B in advance, and the goal is to minimize the number of dropped packets. They show that all greedy protocols have bounded competitive ratio on DAGs. NTG, FTS, and LIS have competitive ratios that are bounded for all networks, while FTG, NTS, SIS have an unbounded competitive ratio on cycles. For the line network, it has been shown that if $B = 1$, any online deterministic algorithm is $\Omega(n)$-competitive while for $B > 1$, a competitive ratio of $O(\sqrt{n})$ can be achieved. For further research using this framework, see for example [1, 7, 9, 13, 14, 16, 23].

For information gathering on a line, all greedy protocols are identical from the point of view of throughput or packet loss. A lower bound of $\Omega(\sqrt{n})$ on the competitive ratio of the greedy protocol was given in [2]. Rosen and Scalosub [23] give tight bounds on the competitive ratio of the greedy algorithm as a function of the injection rate of the adversary and the buffer size B. Their results imply that the greedy policy requires $\Theta(n)$-sized buffers to assure no packet loss. Further studies in lines, rings, and trees, were done by Azar and Zachut [9] and for directed grids in [14, 15].

The papers closest to our work are [21] and [17]. Patt-Shamir and Miller [21] study the same problem as this paper. They consider a more general injection model with injection rate ρ (equal to link capacities) and burstiness bounded by σ. In this model they give a centralized algorithm that achieves information gathering without packet loss using buffers of size $\sigma + 2\rho$ [1] and provide a matching lower bound. The algorithm, called *Forward-If-Empty (FIE)*, is unavoidably centralized, relying on simultaneously forwarding long *trains* of packets. They also analyze several local algorithms and for each of them show that in the worst case the buffer sizes are either unbounded, or at least $\Omega(n)$.

Kothapalli and Scheideler [17] study the competitive ratio of the buffer size achieved by algorithms for the problem of information gathering on an *undirected* path. Their adversarial model is significantly different and much stronger than ours: their adversary can not only choose the site of packet injection, but can also decide which edges are active. They show a lower bound of $\Omega(\log n)$ on buffer sizes, as well as an algorithm which asymptotically matches this bound. Their algorithm forward packets in both directions, and therefore does not work on the directed path. In a follow-up paper [18], the authors show that any deterministic algorithm requires $\Omega(n)$-sized buffers in spider-graphs in the worst case.

[1] Actually, it can be shown that the algorithm as it is formulated in [21] uses for $\rho > 1$ buffers of size $\Omega(\log \rho)$. However, it can be easily corrected by not activating a single path and taking ρ packets along it, but by having ρ activating steps, each applying to a single packet.

1.2 Our results

We start by pointing out that a slight variation of the *Local-Downhill* algorithm, shown in [21] to require buffers of size $\Omega(n)$ in paths, can in fact work with buffers of size $O(\sqrt{n})$, improving upon the other local algorithms presented there.

We then show a tight bound of $\Theta(\log n)$ on the buffer size needed by any local algorithm for information gathering on directed paths and trees. On one hand, we prove a lower bound of $\Omega(c \log n/\ell)$ (more precisely $c(1 + (\log n - 2\log \ell - 1)/2\ell))$ for the buffer sizes required by ℓ-local algorithms on directed paths of length n, where the injection rate of the adversary and the link capacity are both c. The lower bound also holds for bidirectional paths, albeit with a constant factor that is worse by a factor of 4. This is significantly tighter than the result of [17], and it applies to arbitrary constant locality ℓ.

On the other hand, for $c = 1$, we give local algorithms for directed paths and trees that require buffers of size $O(\log n)$. For the directed path, we give a very simple 1-local algorithm that achieves an upper bound of $\log n+3$, i.e. within a factor of 2 of the lower bound. In comparison with the algorithm from [17], our algorithm is simpler, achieves a better bound and works on a directed line, while the algorithm of [17] balances queues by sending packets away from the sink. However, the adversary considered in [17] is stronger, and thus the results are not comparable. For the directed tree, we give a very simple 2-local algorithm that achieves an upper bound of $O(\log n)$. This is in contrast with the lower bound of $\Omega(n)$ shown in [18] for spider graphs, emphasizing the difference between our adversary models.

To the best of our knowledge, the previous best local algorithm for convergecast in trees required buffer size $\Omega(n)$ in the worst case. While our algorithms are very simple to specify and implement, the analysis of both algorithms is based on a sophisticated book-keeping scheme.

After this paper was accepted, we were notified that the same algorithm for paths and trees was independently and concurrently proposed by Patt-Shamir and Rosenbaum [22].

2 NOTATION AND PRELIMINARIES

We consider tree networks of n nodes. The root of the tree, denoted s, is the sink node, which *consumes* packets. The nodes model hosts or routers in a communication network, and the edges represent communication links between them. Each edge can forward at most c packets along every outgoing link in every step. We consider an adversary of rate c; in every time step, the adversary injects a total of at most c packets at some nodes in the network. Our lower bounds work for any c, while our algorithms assume that $c = 1$. Since every packet is to be routed to the sink, the path taken by a packet is assumed to be the unique shortest path to the sink and does not need to be specified. As is common in the literature, we assume that time is divided into steps, each of which can be divided into 2 mini-steps. In the first mini-step, the adversary injects $\leq c$ packets into the network, and can choose the locations for the injections arbitrarily. In the second mini-step, each node uses its scheduling policy to forward at most c packets on each of its outgoing links.

For every node v, we denote by $s(v)$ its *successor* along the path to the sink. The *height* of a node v is the number of packets in its buffer, and is denoted by $h(v)$. A *configuration* C specifies the state of the network at the beginning of a given step. For our purposes, a configuration is specified by the heights of all nodes in the network. We denote the height of a node x in configuration C by $h_C(x)$. We assume that $h_C(s)$ is always 0. Let C be a configuration at the start of a step, and C' be the configuration at the start of the following step. We use shorthands $h(x)$ and $h'(x)$ for $h_C(x)$ and $h_{C'}(x)$, respectively. Throughout the paper, we denote by t the node into which the adversary injected a packet.

3 LOWER BOUNDS

In this section, we show lower bounds on the buffer size of ℓ-local algorithms for information gathering on paths; i.e. a node sees the buffer states of all other nodes up to hop distance ℓ, but not more.

THEOREM 3.1. *Any ℓ-local algorithm for information gathering on a directed path with link capacities c requires buffers of size $\Omega(c \log n/\ell)$.*

PROOF. Let n_0 be the largest number of form $\ell 2^i$ that is smaller than n. The adversary works in stages. At the beginning of stage i, at time t_i, it assumes that there is a contiguous block B_i of nodes of size $K_i = n_0/2^i$ such that the average message density in B_i is at least $H_i = c(1+i/2\ell)$, i.e. the total number of messages M_i in the block B_i is at least $K_i H_i$. We show that as long as $K_i \geq 2\ell$, in $x_i = K_i/2\ell$ steps, the adversary is able to construct a block B_{i+1} of size K_{i+1} and average density H_{i+1}. This implies a lower bound of $\lceil H_{i'} \rceil$, where $i' = \log(n_0/2\ell)$ is the number of stages.

We start by showing that the assumption holds for stage $i = 0$. In each of the first n_0 steps, the adversary injects c messages at the leftmost node of the path. Set the initial block B_0 to be the leftmost n_0 nodes; i.e. $K_0 = n_0$ and $t_0 = n_0$. This yields $H_0 = c$, as none of the messages had time to travel outside block B_0.

Consider now the inductive step i.e. assume the inductive hypothesis holds for stage i. First, consider a scenario in which the adversary injects c messages at the rightmost node of B_i for $x_i = K_i/2\ell$ steps starting at time step $t_i + 1$. As the number of injected messages equals the available outflow from B_i, the number of messages in B_i cannot decrease.

Let M_r and M_l be the number of messages in the right and left half of B_i, respectively, at time $t_{i+1} = t_i + x_i$. By the inductive assumption it holds $M_l + M_r \geq K_i H_i$. If $M_r \geq H_{i+1}K_{i+1} = (H_i + c/2\ell)K_i/2 = H_i K_i/2 + cK_i/4\ell = H_i K_i + cx_i/2$, then the right half of K_i satisfies the condition for stage $i+1$ at time t_{i+1} and we are done. Otherwise, we have $M_l = H_i K_i - M_r \geq H_i K_i/2 - cx_i/2$.

Consider now an alternative scenario, in which the adversary instead injects messages into the leftmost node of B_i. As x_i is chosen in such a way that the information from the

boundary of B_i is not able to reach the middle of B_i in time t_{i+1}, the flow of messages through the middle link is the same in both scenarios. Hence, the number of messages in the left half of B_i is now $M_l + cx_i \geq H_iK_i/2 - cx_i/2 + cx_i = H_{i+1}K_{i+1} + cx_i/2 = H_{i+1}K_{i+1}$.
Therefore, the adversary can always select a scenario in which the assumption for level $i+1$ are satisfied. This argument holds as long as $x_i \geq 1$, i.e. $K_i \geq 2l$. The number of stages is $\log(n_0/2\ell) = \lfloor \log(n/2\ell^2) \rfloor$, resulting in maximal buffer size of at least $c(1 + (\log n - 2\log \ell - 1)/2\ell) \in \Omega(c \log n/\ell)$ □

COROLLARY 3.2. *If the insertion model allows for insertion of c messages with additional burstiness of δ [21], then then the adversary can force buffers of size* $c(1 + (\log n - 2\log \ell - 1)/2\ell) + \delta \in \Omega(c \log n/\ell + \delta)$.

PROOF. The adversary follows the same approach, and in the final stage adds an insertion burst of additional δ messages. □

A natural question is whether giving the algorithm the power to forward messages in both directions might help it to overcome the $\Omega(\log n)$ barrier. We answer this question in the negative and show below that using bidirectional links only reduces the constant factor in the lower bound:

THEOREM 3.3. *Any ℓ-local algorithm for information gathering on an undirected path with link capacities c requires buffers of size* $\Omega(c \log n/\ell)$.

PROOF. Omitted due to lack of space. □

4 1-LOCAL ALGORITHM FOR PATHS

In this section, we give an optimal 1-local algorithm for buffer management that achieves information gathering on a directed path using $\Theta(\log n)$ buffer size, for injection rate and link capacity $c = 1$. Recall that the local algorithms discussed in [21] have either unbounded buffer size (e.g., *FIE*) or use buffers of size $\Omega(n)$ (*Downhill*, *Greedy*). In fact, a simple modification of *Downhill* can be shown to achieve significant improvement to $O(\sqrt{n})$:

THEOREM 4.1. *Consider the local algorithm* Downhill-or-Flat *which forwards a packet whenever the buffer of its successor contains equal or smaller number of packets than its own buffer. Algorithm* Downhill-or-Flat *uses buffers of size* $\Theta(\sqrt{n})$.

PROOF. Omitted. □

Looking at the lower-bound examples given by Miller and Pat-Shamir for various local algorithms, we notice that:

- when the adversary injects at the left, the algorithm should efficiently (at throughput 1) forward messages to the right, otherwise the messages pile up on the left (*FIE* and *Downhill* fail in this). In particular, this suggests forwarding messages to the right if the buffer heights are equal.
- when the adversary injects at the right, the messages should not keep arriving from the left, otherwise they pile up on the right (*Greedy* fails in this, but also *Downhill-or-Flat*).

These two requirements seem contradictory, with no apparent way to satisfy them both. The main idea of our algorithm is to satisfy the first requirement for messages on odd heights, and the second one on even heights. If the adversary starts injecting at the right, the packets start to pile up to the next height, switching to the "stopped" behaviour and spreading the piling up leftwards instead of up. If the adversary starts injecting on the left into stopped even-height nodes, the height raises to even and the packets start efficiently flowing to the right. In this way, the algorithm automatically adapts to the adversary's behaviour. Before having a closer look at the behaviour of Algorithm Odd-Even, let us introduce some notation. Let us call a node an *up* node if its height went up, and a *down* node if its height went down, i.e. $h(x) < h'(x)$ for *up* node x and $h(x) > h'(x)$ when x is a *down* node; the nodes of unchanged height are *steady*. Note that as the link capacity is 1, the height of a *down* node is always reduced by 1, while an *up* node can have its height raised by 1 or by 2 (if it received from its predecessor and from the adversary, but did not send – at any round there can be at most one such node, called 2*up*). There is a special type of *up* node: the node that went up from 0 to 1, while all the nodes in front of it are of height 0. We will call it a *leading-zero* node. Note that there might not be an *leading-zero* node in the network.

Algorithm 1: Algorithm Odd-Even executed by node v

if *If $h(v)$ is odd* **then**
  forward a packet to your successor $s(v)$ iff $h(s(v)) \leq h(v)$
else
  forward a packet to your successor $s(v)$ iff $h(s(v)) < h(v)$
end

Consider first a round in which the adversary did not inject any message. In such case, *up* and *down* nodes must alternate in the sense that the first node in any chain of sending nodes is always *down*, and the first node following this chain is always The injection of a message by an adversary merely raises the height of the injected node by one, e.g. making an *up* node out of a *steady* one, or a 2*up* node out of an *up* one.

4.1 Balanced Matchings

In order to show that the heights of nodes do not go up too much, if the height of a node x goes up, we would like to "charge" that increase to another node y whose height went down in the same round. Intuitively speaking, this is as if y *gave* one if its packets to x.

We say that a non-steady node x is a *neighbour* of a non-steady node y iff there are only *steady* nodes between them.

Definition 4.2. A set P of node pairs is a *balanced matching* for a configuration C' iff

- every *up* node is paired with a neighbouring *down* node, except possibly for the *leading-zero* node
- every *down* node is paired with a neighbouring *up* or 2*up* node, except possibly the rightmost *down* node
- the 2*up* node, if any, is paired with its two neighbouring *down* nodes
- no *steady* node is paired with another node

These possible pairs (and one triple) will be called *down-up*, *up-down* and *down*-2*up*-*down* intervals, based on the type of nodes when traversing from the left. In what follows, the *down*-2*up*-*down* interval will implicitly be treated as a *down-up* interval followed by an *up-down* interval.

Algorithm 2: Creating a Balanced Matching

```
Set X to be the set of non-steady nodes of C', with the
  2up node (if any) treated as two consecutive up nodes.
while X contains at least two nodes do
  | processing from the left, let x and y be the first two
  |   non-steady nodes in X.
  | pair x with y and remove them from X
end
```

Claim 1. *At most one non-steady node remains unmatched after executing Algorithm 2, and it is either the rightmost* down *node, or the* leading-zero*.*

Proof. First, note that Algorithm 2 fails to make *up-down* or *down-up* pairs only if there are three consecutive *down* or *up* nodes. However, this never happens: If there is no injection, the *down* and *up* nodes alternate. If there is an injection at node t, it can either make a *steady* node out of a *down* node, make an *up* node out of a *steady* node or make a 2*up* node out of an *up* node. In any case, as before the injection the *up* and *down* nodes alternated, at most two consecutive *up* nodes are created and there are no two consecutive *down* nodes.

As each iteration of the while loop removes two non-steady nodes, only the rightmost non-steady node remains unmatched, and only in case the number of non-steady nodes was even (counting the 2*up* node as 2 and the *down*-and-injected node as 0). Hence, it remains to be shown that if the remaining node is an *up* node, it must be a *leading-zero*. If there is *leading-zero*, by its definition it is the rightmost *up* node and we are done. Consider the chain of sending nodes ending in the sink. If there is no *leading-zero* node, the neighbour of the sink must be of non-zero height and hence this chain is non-empty. If there is no injection into this chain, the first node of this chain must go down and being the rightmost non-steady node, the lemma holds.

Finally, if there is injection into this chain, as the *down* and *up* nodes alternate for non-injection case, before the injection the number of non-steady nodes was odd (starting with *down* and finishing with *down*). The injection either creates a rightmost *up* node (if inserted inside the chain), which will pair with the *down* node at the beginning of the chain, or it transforms the rightmost *down* node into a *steady* one. In either case, no unpaired non-steady node remains. □

Lemma 4.3. *Algorithm 2 creates a balanced matching.*

Proof. Consider the processing of X in the while loop. As the *up* and *down* nodes alternate, starting with a *down* node, *down-up* intervals are created before encountering the injected node. If an injection creates two neighboring *up* nodes, switching to *up-down* intervals starting at the injected node takes care of all the remaining non-steady nodes, with Claim 1 taking care of the last non-steady node, if there is any. Note that by construction no steady node is paired. □

The pairs of the balanced matching will be called *matching pairs.*

The adversary could conceivably create a high-height node v by first cheaply creating a lot of low-height nodes and then charging those while increasing the height of v; we prevent that be requiring $h_C(y) \geq h_C(x)$. The next lemma shows that this requirement, as well as monotonicity of the intervals between the nodes of the matching pairs, is indeed satisfied:

Lemma 4.4. *Let (x_d, x_u) be a matching pair with x_u being the* up *node of this pair. Then $h(x_u) \leq h(x_d)$.*

Moreover, if (x_d, x_u) is a down-up *interval, then $h(z) \geq h(s(z))$ for all nodes $z \neq x_u$ between x_d and x_u, and if (x_u, x_d) is an* up-down *interval, then $h(z) \leq h(s(z))$ for all nodes $z \neq x_d$ between x_u and x_d.*

Proof. Let us first consider the case of (x_d, x_u) being a *down-up* interval, i.e. x_d is behind x_u. As x_d went down, $x_d \neq t$ and it sent a message to $s(x_d)$), i.e. $h(x_d) \geq h(s(x_d))$. As none of the nodes between x_d and x_u changed their height, each one of them must have received and sent a message[2]. Combining with the fact that in any chain of sending nodes, the node heights are non-increasing yields the lemma.

If (x_u, x_d) is an *up-down* interval, then x_d has sent to $s(x_d)$, but received nothing from its predecessor $pr(x_d)$. If none of the nodes from x_u to $pr(x_d)$ has sent a message, then their heights form a non-decreasing sequence and the lemma holds. However, there cannot be a node x' between x_u and x_d that has sent a message – the first non-steady successor of such a node would be an *up* node, violating the definition of *up-down* interval. □

4.2 Attachment Scheme

If $h(x_d) > h(x_u)$, the adversary pays for raising the height of x_u by lowering the height of a costlier, higher height node, a net loss. However, the case of $h(x_d) = h(x_u)$ allows the adversary to raise a node height without losing the effort invested into another node of higher height. The core of the proof is to show that in Algorithm Odd-Even such a situation cannot occur too often. To accomplish this, when x_u charges to x_d and $h(x_d) = h(x_u)$, we take note that x_d "gave" x_u a

[2]Observe that if a node sends a message and receives injection, it is not included in the balanced matching

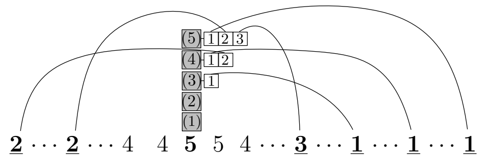

Figure 1: An illustration of a node x of height 5 and all the nodes attached to its packet slots. Only the node heights are shown, and the nodes are depicted from left to right in decreasing order of distance from the sink.The gray boxes correspond to the packets of x, each accompanied with its available slots and attachments. Residues attached to a packet of x appear in boldface and underlined.

packet by *attaching* x_d with the new $h'(x_u)$-th packet of x_u. This attachment will remain until either x_d changes height or until x_u loses its $h'(x_u)$-th packet. In this way, creating a node x of height $h+1$ uses up two nodes of height h: both x itself (as it is no more of height h), but also a node y that attached to x. Taken to its conclusion, this means that creating a node of height h incurs cost exponential in h – provided the attached node is not reused for another *up* node. The key observation (and design goal) is that once a node y gives away a packet and becomes attached, it cannot be charged again by another *up* node as long as this attachment persists. Such a y then becomes useless to the adversary, and we will thus call y a *residue* - it is a leftover that resulted from the creation of a node of higher height.

In order to maintain proper bookkeeping, some attachments may need to be "passed" to other nodes. To keep track of the fact that y virtually gives a packet to x, we create a pointer from x to y. However, when y virtually received this packet from some other node z in a previous step, it would have had a pointer to z; we retain this history by directly creating a pointer from x to z. Intuitively, the greater the height of a node, the more pointers it will have. We formalize all this in the notion of an *attachment scheme* defined below. For $i \geq 3$, a packet $x[i]$ has $i-2$ available *slots* denoted $x[i,1], x[i,2], \ldots, x[i,i-2]$. Note that every slot $x[i,j]$ satisfies $1 \leq j \leq h(x)-2$.

Definition 4.5. An *attachment scheme* A for a configuration C is a set of ordered pairs of the form $(x[i,j], y)$, where $x[i,j]$ is a packet slot and y is a node distinct from x, such that

(1) $j = h(y)$;
(2) each packet slot or node is attached to exactly one element, i.e. for any $(x'[i',j'], y') \in A$ distinct from $(x[i,j], y)$, we have $x[i,j] \neq x'[i',j']$ and $y \neq y'$;

If $(x[i,j], y) \in A$, we say that slot $x[i,j]$ is attached to node y, and similarly that node y is attached to slot $x[i,j]$ (we may also say $x[i,j]$ is the *guardian* of a *residue* y, as will be explained later). We may sometimes write (x,y) instead of $(x[i,j], y)$ when the values of i,j are irrelevant. The node y attached to $x[i,j]$ is denoted $att_A(x[i,j])$.

Figure 1 illustrates a node x with all its available packet slots attached to a node of appropriate height.

LEMMA 4.6. *Let A be an attachment scheme for a configuration C. Let $m := \max_{x\in V} h_C(x)$, and assume $m \geq 3$. Then there are at least $2^{m-2}-1$ distinct nodes that are a residue of A.*

PROOF. For a node $x \in V$, denote by $R'(x)$ the set of residues of A that are attached to a packet slot of x, i.e. $R'(x) = \bigcup_{3\leq i\leq h(x)} \bigcup_{1\leq j\leq i-2} \{att_A(x[i,j])\}$. Define $R(x)$ inductively as follows: if $h(x) \leq 3$, then $R(x) = R'(x)$, and if $h(x) \geq 4$, then $R(x) = R'(x) \cup \bigcup_{x'\in R'(x)} R(x')$. That is, $R(x)$ is the set of residue nodes that are attached to x directly or indirectly. Observe that for any two x and x' with $h(x) = h(x')$, we have $|R(x)| = |R(x')|$.

For an integer p, denote by $r(p)$ the cardinality of $R(x)$ for a node x such that $h(x) = p$. We show that $r(p) = 2^{p-2}-1$. We have $r(1) = r(2) = 0$, since nodes of height 1 or 2 have no available slots for residues. For $p \geq 3$, a node x with $h(x) = p$ has a packet $x[p]$ with each slot $x[p,1], \ldots, x[p,p-2]$ attached to a residue, and there are $p-2$ of those. Also, for each $1 \leq i \leq p-2$, the residue $att_A(x[p,i])$ implies the existence of $r(i)$ other residue nodes. Moreover, the packet $x[p-1]$ implies the existence of $r(p-1)$ residue nodes. Note that by Rule 2 of attachment schemes, no residue is double-counted. Thus, we get $r(p) = p-2+\sum_{i=1}^{p-1} r(i) = p-2+r(p-1)+\sum_{i=1}^{p-2} r(i) = p-2+r(p-1)+r(p-1)-(p-3) = 1+2r(p-1) = 2^{p-2}-1$, when $r(2) = 0$ (the third equality is due to $r(p-1) = p-3+\sum_{i=1}^{p-2} r(i)$). The Lemma follows by setting $p = m$. □

LEMMA 4.7. *Let A be an attachment scheme for a configuration C. Then $\max_{x\in V} h_C(x) \leq \log n + 3$.*

PROOF. By Lemma 4.6, if $m := \max_{x\in V} h_C(x)$, then there are at least $2^{m-2}-1$ nodes that are a residue. Since all these nodes are distinct, we have $2^{m-2}-1 \leq n$, which yields $m \leq \log n + 3$. □

4.3 Maintaining an Attachment Scheme

Due to lack of space, proofs of lemmas are omitted in this subsection.

If for every configuration C, there exists an attachment scheme, it follows from Lemma 4.7 that the height of every node in the path is always upper bounded by $\log n + 3$. We now proceed to show by induction that every configuration indeed admits an attachment scheme. The initial configuration consists of height 0 (i.e. slot-free) nodes, hence it vacuously admits an attachment scheme. If a configuration C admits an attachment scheme, then we will show that the next configuration C' also admits one. The transition from C to C' is done by handling separately and independently the matching pairs of C': We present an algorithm which processes a matching pair $\{x_u, x_d\}$ by changing the heights of its nodes to their new values in C' and rearranging some attachments coincident with x_u or x_d so that an attachment scheme is maintained.

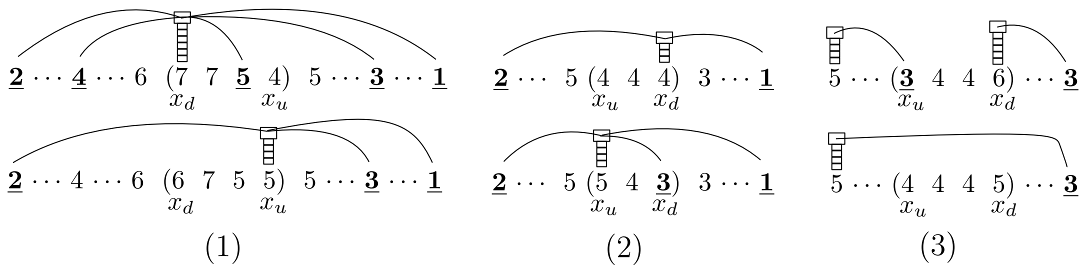

Figure 2: Three examples of applying Algorithm 4. Top: the state before, bottom: the state after. The parentheses surround the processed matching pair. We only represent the packets, attachments and residues of interest. (1) A down-up interval illustrating how x_d passes all possible attachments to x_u (line 7 of Algorithm 4). Note that the residues of value 4 and 5 gets detached in $C_{P'}$. (2) An up-down interval in which $h_d = h_u = 4$. Here x_d passes all its attachments, in addition to becoming a residue attached to x_u (line 9). (3) An up-down interval in which x_u was a residue attached to some slot $z[i, h_u]$ (here z is the node of value 5), and $x_d[h_d, h_u]$ is attached to a node y (y is the node of value 3). After processing, y is attached to $z[i, h_u]$ (line 18).

In order to carry out the inductive step, we need to strengthen the definition of an attachment scheme:

Definition 4.8. An attachment scheme A is *valid*, if in addition to Rules 1 and 2, the following rules are satisfied for each residue y and its guardian x of A:

(3) if $h(y)$ is even, then x is in front of y;
(4) if $h(y)$ is odd, then x is behind y;
(5) for every node z on the path between x and y, $h(z) \geq h(y)$.

Let P be a subset of matching pairs of a balanced matching of C'. We say that C_P is an *intermediate configuration* for P iff $\forall x, x \in P : h_{C_P}(x) = h(x)$, while $\forall x, x \notin P : h_{C_P}(X) = h'(x)$.

We will need the following three technical lemmas:

Lemma 4.9. *If (x_u, x_d) is a matching pair with $h(x_u) = h(x_d) = h$ then x_u is not a residue.*

The following lemma is crucial in proving that the residues are not shared:

Lemma 4.10. *Let y be a residue of A. Then y is not a* down *node.*

Lemma 4.11. *The following facts hold when Algorithm 4 processes a matching pair (x_d, x_u):*

(1) *after being processed, no* up *node remains a residue of another node*
(2) *no existing slot has become empty*
(3) *no new empty slot has been created*
(4) *whenever an attachment to residue y is transferred*
  a) *from x_d to x_u on line 7*
  b) *from z to x_d on line 15*
  c) *from x_d to z on line 18*
  $h'(w) \geq h'(y)$ holds for all nodes between the nodes transferring the attachment (endpoints included)
(5) *the relative order (in front of, or behind) between residues and their guardians never changes*

Algorithm 3: Processing a balanced matching.

Input : Configurations C and C' and an attachment scheme A for C
Output : An attachment scheme A' for C'', where C'' differs from C' (and equals C) only for the possible *down*-2*up*-*down* triple, the *leading-zero* and the unmatched rightmost *down* node

```
Let M be a balanced matching for C'
Set P := M and A' := A
while P ≠ ∅ do
    Let (x_d, x_u) be a matching pair from P
    Set A' := processPair(C_P, A', x_d, x_u);
    Set P := P \ {x_d, x_u};
end
Return (A')
```

This allows us to prove that all the rules of the attachment scheme are satisfied:

Lemma 4.12. *Processing one pair by Algorithm 4 maintains all the rules of the attachment scheme.*

We can now prove the upper bound on buffer sizes.

Theorem 4.13. *Algorithm Odd-Even uses buffers of size at most $\log n + 3$.*

Proof. It follows from Lemma 4.12 that processing all pairs of a balanced matching by Algorithm 4 (including the two pairs concerning *down*-2*up*-*down* interval) maintains a valid attachment scheme. What remains to be dealt with is the right-most *down* node and the *leading-zero* node. The last one is not a problem, as it was of height 0 and hence not a residue, nor does it have a packet slot, as it is of height 1. The right-most *down* node could have only released some attachments, and did not gain any, so it does not need any sophisticated (creation/passing) attachment processing (by Lemma 4.10 it was not a residue, so no empty slots were created either).

Algorithm 4: Handling a matching pair.

function *processPair* (C_P, A_P, x_d, x_u);
Input : An intermediate configuration C_P, an attachment scheme A_P for C_P, and a matching pair $(x_d, x_u) \in P$ with x_d and x_u being the *up* and *down* nodes, respectively.
Output : An attachment scheme $A_{P'}$ for $C_{P'}$, where P' is obtained from P by removing (x_d, x_u).
Let $h_d := h(x_d)$ and $h_u := h(x_u)$
Let $A' := A_P$.
if *there is a slot* $x_d[i, h_u]$ *such that* $(x[i, h_u], x_u) \in A'$ *and* $i \neq h_d$ **then**
    Swap the $x_d[i, h_u]$ and $x_d[h_d, h_u]$ attachments: in A', replace $(x_d[i, h_u], x_u)$ by $(x_d[i, h_u], att_{A_P}(x_d[h_d, h_u]))$ and replace $(x_d[h_d, h_u], att_{A_P}(x_d[h_d, h_u]))$ by $(x_d[h_d, h_u], x_u)$. ; `// Here we ensure that when` x_u `gets detached, it does not leave slot` $x_d[i, h_u]$ `empty`
end
Pass all possible attachments from the $x_d[h_d]$ packet to the $x_u[h_u + 1]$ packet and remove the others, i.e. remove from A' all the attachments $\{(x_d[h_d, i], att_{A_P}(x_d[h_d, i])) : 1 \leq i \leq h_d - 2\}$ and add to A': $\{(x_u[h_u + 1, j], att_{A_P}(x_d[h_d, j])) : 1 \leq j \leq \min(h_d - 2, h_u - 1)\}$
if $h_d = h_u$ *and* $h_d \geq 2$ **then**
    Add $(x_u[h_u + 1, h_u - 1], x_d)$ to A'
end
if x_u *is a residue of* A_P **then**
    Let $z[i, h_u]$ be the packet slot attached to x_u in A'
    Remove the $(z[i, h_u], x_u)$ attachment from A'
    if $h_d = h_u + 1$ **then**
        Add to A' the attachment $(z[i, h_u], x_d)$
    else if $h_d \geq h_u + 2$ *and* $z \neq x_d$ **then**
        Let $y = att_{A'}(x_d[h_d, h_u])$
        Add to A' the attachment $(z[i, h_u], y)$

end
Return A' as $A_{P'}$

Note that handling the *down*-2*up*-*down* interval as a sequence of two intervals sharing an *up* node is perfectly fine: from the point of the right pair this looks the same as if t was of height $h(t) + 1$ and received a message from the left. Lemma 4.7 now completes the proof of the theorem. □

5 2-LOCAL ALGORITHM FOR TREES

Notice that in this section, due to lack of space, all proofs are omitted.

The first observation is that lookahead of 1 is not sufficient: Consider node u having $\sqrt{n}$ neighbours and the same schedule as discussed in its caption. When the packets arrive simultaneously to v's, each v_i will send a packet to u, forcing u to need buffer of size $\sqrt{n}$.

Hence, we consider a 2-local algorithm. The algorithm is a straightforward generalization of Algorithm Odd-Even:

Algorithm 5: Algorithm Tree

if *the height* h *of the node is odd* **then**
    forward a packet to your successor iff its height is at most h and you have the highest priority among your siblings
else
    forward a packet to your successor iff its height is less than h and you have the highest priority among your siblings`// even height` h
end

The algorithm is completed by specifying the priority scheme: A sibling with a higher height has higher priority. Among the siblings of the same maximal height, choose arbitrarily.

Let us now introduce more nomenclature. An internal node v of in-degree at least 2 will be called an *intersection*. For a fixed round, in each intersection there will be at most one incoming packet; the branch where it comes from will be called a *priority line*[3]. A non-priority line ends in a *blocked* node. Hence, the tree can be viewed as a set of lines, starting in leaves and ending in blocked nodes, with one branch, called *drain* making it all the way to the sink. One of the lines might contain the injected node – we will call it the *injected* line. All other lines are *normal*. Note that the *up* and *down* nodes on non-injected lines alternate, starting with a *down* node (exactly like in paths) and ending with a *leading-zero* or *down* node if the line is a *drain*, otherwise ending with an *up* node.

Algorithm 6: Balanced Matching on a Tree

For each line, apply the balanced matching algorithm for paths:
if *the injection was on the priority line to the sink* **then**
    we are done, nothing left to do
else
    while *there is an unmatched* up *node* x_u **do**
        Let v be first intersection in front of x_u, and let p_v be the priority line containing v.
        Let x_d be the first *down* node behind v on p_v.
        Remove the pairs (including the one containing x_d) in front of x_d on the line of x_d
        Add (x_d, x_u) to the set of matching pairs
        Process the remainder of the x_d's line using the algorithm for paths (i.e. add up-down pairs while possible)
    end
end

[3]It can happen that the intersection has no incoming packet. In such a case, we choose as the priority line, the line into which there was an injection; if no such line is behind, select arbitrarily.

The first step is a generalization of balanced matchings to trees: The matchings for each normal line translate directly (they are each just a collection of *down-up* intervals from left to right). If the injected line is also the *drain* one, this is handled as in the single line case with injection. But if the injected and *drain* lines are different, let v be the intersection on which the injected line blocks. As each normal blocked line has an equal number of *up* and *down* nodes, the injected line has an excess of one *up* node: Applying Algorithm 2 leaves it with the rightmost *up* node x unpaired. At this moment, it is impossible to carry on constructing balanced matching as a union of balanced matchings of the lines: We need to introduce *crossover pairs* containing nodes from different lines. This is what is done in the while loop: As the last non-steady node y of the priority line is *down*, we pair x with y to form a *crossover pair*. Since we have removed y from its line, we need to re-do its pairings that were in front of y, switching to *up-down* intervals. This possibly leaves another unmatched *up* node at the end, which needs to be handled in the same manner. We make these crossover pairs until we eventually reach the drain, where the *up-down* matchings do not leave an unmatched *up* node at the end. An example of applying Algorithm 6 is shown in Figure 3. Hence, a tree-version of Lemma 4.3 holds:

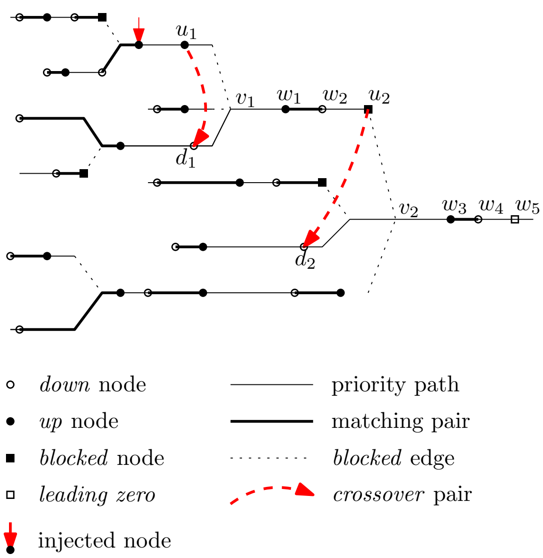

Figure 3: Constructing balanced matching on a tree. Due to adding the (u_1, d_1) matching, the matchings $(d1, w1)$ and $(w2, u2)$ were removed and replaced by $(w1, w2)$, leaving u_2 unpaired. This forced the (d_2, u_2) matching, then switching the (d_2, w_3) and (w_4, w_5) into (w_3, w_4) and leaving w_5 unpaired.

Lemma 5.1. *Algorithm 6 creates a balanced matching.*

We will often make use of the following simple property of matching pairs.

Lemma 5.2. *Let x_u be an* up *node lying on a priority path p that is not the drain, and let v be intersection node on which p does not have priority. Then x_u is matched with a node x_d behind v.*

In paths, the notion of *between* two nodes is straightforward. In trees, we will generalize it to fit our purpose: *between* x and y is satisfied by all nodes on the path from x to y, *except* for the node v (if any) in which this path changes direction from forward to backward. This node will be called the *tip* of the crossover pair.

Before introducing the tree-version of Lemma 4.4 we need a bit more notation: Let (x, y) be a crossover pair with tip v. $p_v(z)$ will denote the predecessor of v on the path from z to v. If clear from the context, we will omit the subscript v.

We now show a tree-version of Lemma 4.4:

Lemma 5.3. *Let (x_d, x_u) be a matching pair with x_u being the* up *node of this pair. Then $h(x_u) \leq h(x_d)$ and $h(z) \geq h(x_u)$ for all nodes z between x_u and x_d.*

Moreover, the nodes on the path from x_d to x_u appear in non-increasing order of height, with the possible exception of the tip v between x_d and x_u.

The attachment scheme is defined analogously as for the path case. However, in order to limit technicalities, we limit Rule 2 to residues of even value. This implies that Lemmas 4.6 and 4.7 yield a $2 \log n + O(1)$ bound.

The Rules 3, 4 and 5 are replaced as follows:

Definition 5.4. For each pair (x, y) of an attachment scheme, where y is a residue and x is its guardian, the following rules must be satisfied:

(6) if $h(y)$ is even, x is not behind y;
(7) if (x, y) is not a crossover pair, then $h(z) \geq h(y)$ holds for every node z on the path between y and $p(y)$; otherwise if (x, y) is a crossover pair, $h(z) \geq h(y)$ holds for every node z on the path between y and $p(y)$, and $h(z) > h(y)$ holds for every node z on the path between x and $p(x)$.

This allows us to prove (using the same arguments; note that the proof is not valid for odd-height residues) the tree-version of Claim 4.10:

Claim 2. *Let x be an even-height residue of A. Then x does not go down.*

In the rest of the proof, when we discuss residues and attachments, we limit ourselves to even height residues and corresponding attachment pairs.

First, we show that Lemma 4.9 holds also for trees. As this was the only necessary ingredient for Fact 2, this implies that after running Algorithm 4 on every matching pair, the resulting attachment scheme is still full.

Lemma 5.5. *If (x_u, x_d) is a matching pair with $h_u = h_d = h$, then x_u is not a residue.*

The proofs of Facts 1, 2 and 3 of Lemma 4.11, as well as the proofs from Lemma 4.12 that Rules 1 and 2 are satisfied are based on the behaviour of Algorithm 4, using in addition only Claim 4.10 and Lemma 4.9; using Claim 2 and Lemma 5.5 instead, the same proofs apply to trees without need for any modifications.

We prove that, after running Algorithm 4 on a single matching pair, crossover or not, Rules 6 and 7 are satisfied directly (here we do not refer to Facts 4 and 5). As before, $h(x)$ is the height of a node at the start of the round, and $h'(x)$ its height after the round.

We first establish that unmodified attachments are still valid, then proceed with the new attachments created by the algorithm.

LEMMA 5.6. *Let (x, y) be an attachment of A that has not changed after running Algorithm 4 on a matching pair. Then (x, y) still satisfies Rules 6 and 7.*

LEMMA 5.7. *Let (x_u, x_d) be a new attachment created on line 9. Then (x_u, x_d) satisfies Rules 6 and 7.*

LEMMA 5.8. *Let (x_u, y) be an attachment formed by passing y from x_d to x_u on line 7 of Algorithm 4. Then (x_u, y) satisfies Rules 6 and 7.*

LEMMA 5.9. *Let (z, x_d) be an attachment formed by swapping the residue of z from x_u to x_d on line 15 of Algorithm 4. Then (z, x_d) satisfies Rules 6 and 7.*

LEMMA 5.10. *Let (z, y) be an attachment formed on line 18 of Algorithm 4. Then (z, y) satisfies Rules 6 and 7.*

We have shown that after running Algorithm 4 on a given matching pair (x_d, x_u), all the unmodified attachments are still valid, and the newly created ones also satisfy the required rules. As before, after processing every single matching pair, we reach the final configuration along with a full attachment scheme. As the handling of the possible *leading-zero*, *down-2up-down* intervals, and unpaired rightmost *down* node is the same as for paths, this completes the proof that a full attachment scheme is maintained in trees. Combining with Lemmas 4.6 and 4.7 yields:

THEOREM 5.11. *Algorithm Tree uses buffers of size at most $O(\log n)$.*

6 CONCLUSIONS

We studied the information gathering problem in paths and trees under the assumption of adversarial traffic. Given an adversary that can inject at most c packets into the network in every step, we showed an $\Omega(\log n)$ lower bound on the buffer space needed to ensure no packet loss. For $c = 1$, we gave deterministic local algorithms that match this bound for directed paths and trees. The existence of local algorithms with $O(\log n)$ buffers for higher rate adversaries remains open. A natural question to ask is if our algorithms generalize to arbitrary routing patterns, or to DAGs. Another intriguing direction for further research is the delay characteristics of our algorithm as well as those of other algorithms proposed in the literature (for example [17]).

REFERENCES

[1] W. Aiello, Y. Mansour, S. Rajagopolan, and A. Rosén. 2005. Competitive Queue Policies for Differentiated Services. *Journal of Algorithms* 55, 2 (2005), 113–141.

[2] W. Aiello, R. Ostrovsky, E. Kushilevitz, and A. Rosén. 2003. Dynamic routing with fixed size buffers. In *Proceedings of SODA*. 771–780.

[3] C. Alvarez, M. Blesa, and M. Serna. 2004. A characterization of universal stability in the adversarial queueing model. *SIAM J. Comput.* 34, 1 (2004), 41–66.

[4] M. Andrews. 2004. Instability of FIFO in session-oriented networks. *Journal of Algorithms* 50, 2 (2004), 232–245.

[5] M. Andrews, B. Awerbuch, A. Fernández, T. Leighton, Z. Liu, and J. Kleinberg. 2001. Universal-stability Results and Performance Bounds for Greedy Contention-resolution Protocols. *J. ACM* 48, 1 (Jan. 2001), 39–69.

[6] M. Andrews, A. Fernandez, A. Goel, and L. Zhang. 2005. Source routing and scheduling in packet networks. *J. ACM* 52, 4 (2005), 582–601.

[7] S. Angelov, S. Khanna, and K. Kunal. 2009. The network as a storage device: dynamic routing with bounded buffers. *Algorithmica* 55 (2009), 71–94.

[8] E. Anshelevich, D. Kempe, and Kleinberg. J. 2008. Stability of load balancing algorithms in dynamic adversarial systems. *SIAM Journal of Computing* 37, 5 (2008), 1656–1673.

[9] Y. Azar and R. Zachut. 2005. Packet routing and information gathering in lines, rings, and trees. In *Proceedings of ESA*. 484–495.

[10] R. Bhattacharjee, A. Goel, and Z. Lotker. 2005. Instability of FIFO at arbitrarily low rates in the adversarial queueing model. *SIAM J. Comput.* 34, 2 (2005), 318–332.

[11] A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. P. Williamson. 2001. Adversarial Queuing Theory. *J. ACM* 48, 1 (2001), 13–38.

[12] J. Diaz, D. Koukopoulos, S. Nikoletseas, M. Serna, P. Spirakis, and D. Thilikos. 2001. Stability and non-stability of the FIFO protocol. In *Proceedings of SPAA*. 48–52.

[13] G. Even and M. Medina. 2010. An $O(\log n)$-Competitive Online Centralized Randomized Packet-Routing Algorithm for Lines. In *Proceedings of ICALP, Part II*. 139–150.

[14] G. Even and M. Medina. 2011. Online packet routing in grids with bounded buffers. In *Proceedings of SPAA*. 215–224.

[15] G. Even and M. Medina. 2016. Online packet routing in grids with bounded buffers. *Algorithmica* (2016). https://doi.org/10.1007/s00453-016-0177-0

[16] E. Gordon and A. Rosén. 2005. Competitive Weighted Throughput Analysis of Greedy Protocols on DAGs. In *Proceedings of PODC*. 227–236.

[17] K. Kothapalli and C. Scheideler. 2003. Information gathering in adversarial systems: lines and cycles. In *Proceedings of SPAA*. 333–342.

[18] K. Kothapalli and C. Scheideler. 2006. Lower bounds for information gathering in adversarial systems. In *Proceedings of International Conference on Distributed Computing in Sensor Systems*.

[19] D. Koukopoulos, M. Mavronicolas, S. Nikoletseas, and P. Spirakis. 2002. On the stability of compositions of universally stable, greedy contention-resolution protocols. In *Proceedings of DISC*. 88–102.

[20] Z. Lotker, B. Patt-Shamir, and A. Rosén. 2004. New stability results for adversarial queueing. *SIAM J. Comput.* 33, 3 (2004), 286–303.

[21] A. Miller and B. Patt-Shamir. 2016. Buffer Size for Routing Limited-Rate Adversarial Traffic. In *Proceedings of Distributed Computing: 30th International Symposium, DISC 2016*. 328–341.

[22] B. Patt-Shamir and W. Rosenbaum. 2017. The Space Requirement of Local Forwarding on Acyclic Networks. In *Proceedings of PODC, to appear*.

[23] A. Rosén and G. Scalosub. 2007. Rate vs. Buffer Size: Greedy Information Gathering on the Line. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures ((SPAA))*. 305–314.