Exploiting Locality in Distributed SDN Control

Stefan Schmid
TU Berlin & T-Labs, Germany
stefan@net.t-labs.tu-berlin.de

Jukka Suomela
Helsinki Institute for Information Technology HIIT
Department of Computer Science
University of Helsinki
jukka.suomela@cs.helsinki.fi

ABSTRACT

Large SDN networks will be partitioned in multiple *controller domains*; each controller is responsible for one domain, and the controllers of adjacent domains may need to communicate to enforce global policies. This paper studies the implications of the local network view of the controllers. In particular, we establish a connection to the field of local algorithms and distributed computing, and discuss lessons for the design of a distributed control plane. We show that existing *local* algorithms can be used to develop efficient coordination protocols in which each controller only needs to respond to events that take place in its local neighborhood. However, while existing algorithms can be used, SDN networks also suggest a new approach to the study of locality in distributed computing. We introduce the so-called *supported locality model* of distributed computing. The new model is more expressive than the classical models that are commonly used in the design and analysis of distributed algorithms, and it is a better match with the features of SDN networks.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; F.1.1 [**Computation by Abstract Devices**]: Models of Computation

Keywords

local algorithms, software defined networking

1. INTRODUCTION

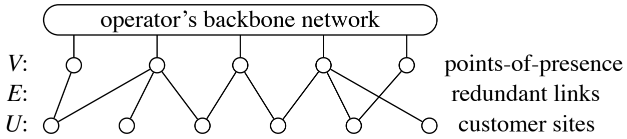
The paradigm of **software defined networking** (SDN) advocates a more centralized approach of network control, where a controller manages and operates a network from a *global* view of the network. However, the controller may not necessarily represent a single, centralized device, but the control plane may consist of multiple controllers in charge of managing different administrative domains of the network or different parts of the flow space. The problem of managing a network and enforcing policies in such a distributed control plane, however, exhibits many similarities with the task of designing **local algorithms**—a well-studied subfield in the area of distributed computing. Local algorithms are distributed algorithms in which each device only needs to respond to events that take place in its local neighborhood, within some constant number of hops from it; put otherwise, these are algorithms that only need a constant number of communication rounds to solve the task at hand.

This paper highlights that there is a lot of potential for interactions between the two areas, the distributed control of SDN networks and local algorithms. On the one hand, we argue that there are many recent results related to local algorithms that are relevant in the efficient management and operation of SDN networks. On the other hand, we identify properties of SDN networks which raise additional and new challenges in the design of local algorithms. We describe a disparity between the features of SDN networks and the standard models of distributed systems that are used in the design and analysis of local algorithms. Indeed, there are many tasks that can be solved efficiently in real-world SDN networks, yet they do not admit a local algorithm in the traditional sense. We suggest a new model of distributed computing that separates the relatively static network structure (e.g., physical network equipment) and dynamic inputs (e.g., current traffic pattern).

1.1 Running Examples

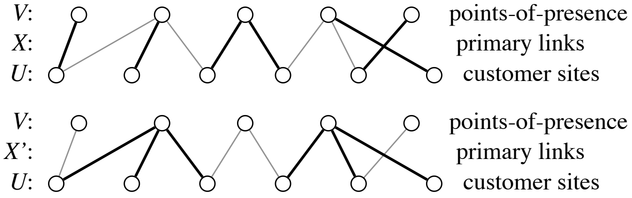
We begin our exploration of the interactions between distributed SDN control and local algorithms with two scenarios that we will use as running examples.

Example 1. Link assignment. Consider an Internet Service Provider with a number of Points-of-Presence $v \in V$, and a number of customers $u \in U$. For each customer, there are multiple redundant connections between the customer's site and the operator's network. We can represent the connections between the customer sites and the access routers in the operator's network as a bipartite graph $G = (U \cup V, E)$, where an edge $\{u, v\} \in E$ indicates that there is a network link from customer site u to the access router in point-of-presence v.



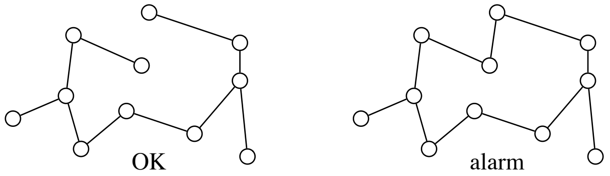
HotSDN'13, August 16, 2013, Hong Kong, China.

The task is to select the links for the primary connection between the customer's site and the Internet: that is, we have to activate one link for each customer. More formally, we need to select a subset of edges $X \subseteq E$ such that each customer site is incident to one edge in X; see below for two examples.



Not all solutions are equally good. While the solution X above provides a fairly nice load balancing in the operator's network, solution X' is much worse: some routers are idle while others are serving a large number of customers. Our goal is to find a *balanced* assignment, automatically and quickly—we want to respond to changes in the network (e.g., a primary link going down) as fast as possible.

Example 2. Spanning tree verification. There are many protocols that aim at *constructing* a spanning tree (e.g., the *STP protocol* in the context of layer-2 networking). In this example scenario, we consider the seemingly simpler task of *verifying* the correctness (e.g., loop-freeness) of a spanning tree (or forwarding set). We are given a spanning tree—constructed by some other protocol—and our goal is to detect errors as efficiently as possible. More precisely, we want to raise an alarm (e.g., trigger a re-computation of the spanning tree) whenever we notice that the tree is disconnected or contains loops (e.g., in the forwarding set).



1.2 Distributed Control in SDN

At first sight, both of our examples seem to be straightforward to solve in the SDN paradigm, given a single global controller with an up-to-date network view: In the link assignment example, the controller is aware of the structure of graph G; the controller can use an algorithm that finds an optimal assignment, and configure the switches/routers accordingly. Also the spanning tree verification is trivial—indeed, if the controller itself constructs the spanning tree, no verification is needed. The controller has a global view of the network, and it can easily ensure that the network is in a consistent, optimal configuration.

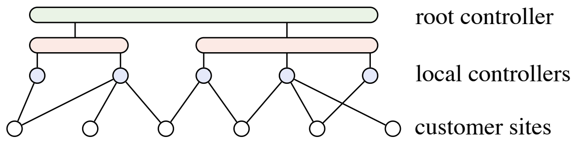
However, the situation changes once control becomes distributed, and there are many reasons to use a distributed control plane [11]: (1) administration; (2) SDN providers with a geographically local footprint; (3) scalability; (4) reducing the latency from a switch to its closest controller; (5) fault-tolerance; and (6) load-balancing. Indeed, efficiency is an important motivation besides administration and fault-tolerance: even within a single administrative domain, multiple controllers may be used for offloading computational tasks (see, e.g., the Kandoo framework [9]).

Hence we have to cope with a network of distributed controllers; each controller has a partial view of the network, and the controllers have to collaborate and exchange information with each other.

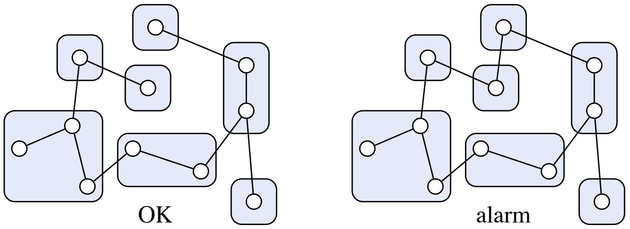
1.3 Structure of the Control Plane

We can identify two main types of distributed control control planes: a *flat* control plane, in which the controllers partition the network into disjoint areas (horizontal partition), and a *hierarchical* control plane, in which the functionalities of the controllers are organized vertically. In the flat control plane, the structure is often given by administrative constraints (e.g., each part corresponds to the network controlled by one organization), whereas in a hierarchical control scenario, we can select an appropriate structure.

Hierarchical SDN Control. Tasks are distributed to different controllers, e.g., depending on the locality requirements: while certain tasks such as heavy-hitter detection may be performed topologically close to an SDN switch, other tasks such as routing require a more global network view. An example of a hierarchical control is Kandoo [9], which organizes controllers into layers. At the bottom layer only local control applications are run (i.e., applications that can function using the state of a single switch). The local controllers handle most of the frequent events near the data path and shield the higher layers, which can hence focus on more global events The following figure illustrates the structure of a hierarchical SDN network in the context of the link assignment task (Example 1 with two levels).

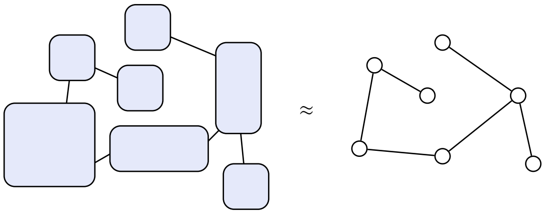


Flat SDN Control. The network is partitioned into different controller domains, where each controller manages a disjoint set of SDN-enabled switches. The structure is mainly motivated by administrative constraints: different economical players own different parts of the network and manage them more or less independently. Another reason for a flat distributed control may be to reduce the latency between switches and controllers, and to keep the control traffic local (e.g., Heller et al. [10]). In both cases, the SDN switches managed by a given controller are most likely connected (local "topological footprint"), but they may not have a geographically local footprint (see, e.g., wide-area SDN networks [15]). Below we have a schematic illustration of the structure of a flat SDN network in the context of the spanning tree verification task (Example 2).



1.4 Implications of Distributed Control

Our running examples are no longer trivial to solve efficiently if we have a distributed control plane. The controllers will need to exchange information with each other (directly or indirectly, e.g., via some middleware [4]) in order to solve the task at hand. Especially in the case of a flat SDN control, the *high-level* structure (i.e., network of controller domains, henceforth called *controller graph*) of the SDN control plane network resembles the structure of a traditional network. The following figure illustrates this concept:



2. LOCALITY

Given that the control plane becomes distributed, it is time to revisit the concepts from distributed computing: which lessons are applicable to the practical challenges of the design, management, and operation of SDN networks? In this article, we will mainly focus on the aspect of *locality*. To understand the concept of locality in the context of distributed SDN control, let us begin with a more straightforward setting: locality in traditional communication networks.

2.1 Traditional Networks and Locality

The control plane in a traditional network is best seen as an instance of a classical distributed system. Each device is a computational entity; initially, it is only aware of its immediate surroundings (e.g., its own identity and the state of each network link connected to it), but it can gather more information about the structure of the network by exchanging messages with its neighbors.

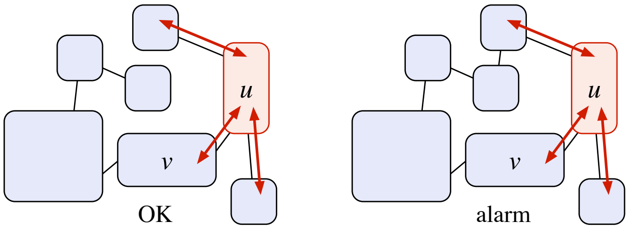
In general, *distances* in the network are related to *costs* (e.g., latency, communication, synchronization, network traffic, coordination among controllers, etc.). If we abstract the cost so that one unit corresponds to the interaction between adjacent control domains (e.g., round-trip time), then at a cost of 1 unit each device can gather information from its distance-1 neighborhood (i.e., its immediately adjacent controller domains). To propagate information further in the control plane, controllers need to communicate repeatedly with their neighborhoods, so that after t iterations (and t units of cost), each device can know (at best) everything about its distance-t neighborhood. In particular, if we need to respond to changes in the network that happened t (logical) "hops" away from us, even in the best case this costs t units. This imposes a fundamental limitation on the controllability of the network.

A key observation is that not all tasks related to network control are alike. Some tasks are inherently *global*: to solve the task, it is absolutely necessary that we respond to events that take place arbitrarily far from us, no matter how clever algorithms and protocols we use. However, some tasks are *local*: to solve the task, it is sufficient that each device only responds to events that take place in its vicinity. More precisely, a task is local if it can be solved with a *local algorithm*; in a local algorithm each node only responds to events that take place within distance t for some constant $t = O(1)$, independently of the size of the network [17, 19].

We will see in Section 3 that link assignment (Example 1) is a local task, while spanning tree verification (Example 2) is a global task.

2.2 Flat SDN Control and Locality

As we have already hinted, in the case of a flat SDN control, the high-level structure of the control plane can be interpreted as a distributed system. More formally, we can construct the *controller graph*, in which each node is a controller, and nodes u and v are connected if the corresponding controller domains of u and v are adjacent.



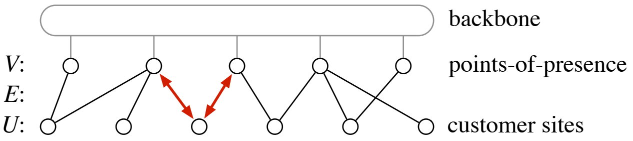
In essence, we abstract away the structure of the underlying physical network (and individual devices in the data plane); following the SDN paradigm, the active elements of the network are controllers, which exchange information with their neighbors.

Often the task at hand also has a natural interpretation from the perspective of the controller graph. This is the case for the spanning tree verification task as well: if each controller ensures that the network topology within its own domain is loop-free and connected (and there are no multiple connections between adjacent controllers), then it is sufficient to verify that the controller graph itself is loop-free and connected.

Now we can take any distributed algorithm that solves the task, and apply it in the controller graph. In particular, if the original algorithm was local, then we would have a very efficient protocol for the operation of the SDN control plane as well.

2.3 Locality-Preserving Simulation

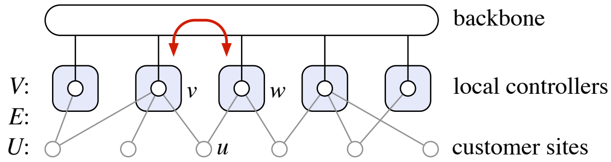
Let us now have a look at the link assignment example. A problem instance is defined by a bipartite graph $G = (U \cup V, E)$ that captures the connections between the customer sites $u \in U$ and the points-of-presence $v \in V$. Therefore—from the perspective of distributed algorithms—it is natural to design algorithms that work in the graph G. Nodes in U and nodes in V are active elements that take part in the execution of the algorithm, and they exchange messages with each other.



While this makes perfect sense in the context of algorithm theory, it looks unwieldy from the perspective of networking: interpreted literally, the network equipment in the operator's network would exchange messages through customer sites.

Fortunately, a much better approach exists: we can *simulate* any algorithm designed for graph G efficiently. For the sake of concreteness, consider a large-scale SDN network that is structured as follows: in the operator's network we have one *local controller* in each point-of-presence $v \in V$. The controller maintains a full information of all network equipment in the point-of-presence, as well as the connections between the network equipment and the customer sites.

Now assume that we have a distributed algorithm A that is designed to be executed in the bipartite graph G; for simplicity, assume that A is a deterministic algorithm (no randomness). We can efficiently simulate the operation of A as follows: Only nodes $v \in V$ participate in the execution of the algorithm; these nodes correspond to the SDN controllers in our network. Each controller $v \in V$ simulates the actions of all customer sites $u \in U$ that are adjacent to v. For example, assume that in the original algorithm, node $v \in V$ sends a message to an adjacent node $u \in U$, which then sends a message to an adjacent node $w \in V$; in the simulation, it is sufficient that the local controller $v \in V$ sends a message to the local controller $w \in V$. All communication takes place through the operator's backbone network.



While such a simulation is always possible, it is important to note that in our example it also preserves *locality*. From the perspective of shortest-path distances in graph G, two points-of-presence $v, w \in V$ are within distance 2 from each other if and only if there is a common customer site $u \in U$ that is connected to both of them. This typically implies that the geographic locations of v and w are relatively close to each other, and therefore it is likely that there is also a short path in the operator's backbone that connects v and w. In particular, if algorithm A is local (in terms of graph G), then each SDN controller only needs information that is near it (in terms of shortest-path distances in the operator's backbone).

2.4 Hierarchical SDN Control and Locality

We can apply the simulation technique in the case of hierarchical SDN networks in a straightforward manner. If we have a controller hierarchy in which the local controllers that are located at each point-of-presence, we can use the approach of Section 2.3 directly: The local controllers are the active participants. Higher-level controllers in the hierarchy do not take part in the execution of the distributed algorithm A; they are only responsible for maintaining the routing tables in the backbone network.

3. FROM LOCAL ALGORITHMS TO SDN

Now that we have seen how to apply local algorithms in the context of SDN networks, let us have a look at some concrete examples.

3.1 Link Assignment

The link assignment task (Example 1) can be formalized as a *semi-matching* problem [8]. If a customer $u \in U$ is connected to site $v \in V$, and there are c customers in total who are connected to the same site (and hence compete of the same limited resources), we say that the *cost* of customer u is c. In the semi-matching problem, the task is to minimize the *average* cost of the customers (in essence, we prefer an assignment in which each customer is connected to a site with few other customers).

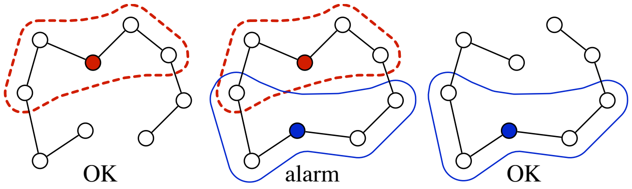
As we now have the task defined in a formally precise manner, we can investigate it from the perspective of locality. It is fairly easy to see that if we want to find an *optimal* assignment, the task is inherently global (consider a graph G that consists of a long path).

However, if we are happy with a *near-optimal* assignment (a constant-factor approximation of the optimum), it turns out that task becomes local; see Czygrinow et al. [2] for a distributed algorithm that solves precisely this task. (Concretely, the running of their algorithm depends on the maximum degree of the bipartite graph G, i.e., the maximum number of links per customer and links per point-of-presence. However, the running time is independent of the *size* of graph G, i.e., the total number of customers or the total number of points-of-presence.)

This is just one example of a local algorithm that can be applied in resource allocation tasks—local algorithms are also known for many closely related problems, such as maximal matchings [7], stable matchings [3], and fractional matchings [14].

3.2 Spanning Tree Verification

While there is an abundance of positive results related to local algorithms and the link assignment task, it is easy to see that spanning tree verification (Example 2) is an inherently global task. If each device is only permitted to gather information in its local neighborhood, feasible spanning trees are indistinguishable from graphs with loops or disconnected graphs. To see this, consider the following examples.



In the above examples, all nodes in the bad instance have local neighborhoods that look identical to the local neighborhoods of at least one good instance. However, one of the nodes in the bad instance has to raise an alarm—if we attempt to do this based on local information only, we will occasionally make false alarms.

Even though the task is inherently global, we can still use local algorithms if we resort to *proof labeling schemes* [6, 12, 13]. If we are given just an arbitrary spanning tree T, we cannot verify it locally. However, it is possible to label tree T with a *locally checkable* proof. The labels are compact—we do not need to waste much storage or bandwidth—yet they contain sufficient information so that we can use a *local* algorithm to verify that tree T is indeed *globally* consistent.

Example 3. We can construct a compact locally checkable proof for a spanning tree T as follows [13]. We pick one node r as the *root* node of tree T. Then each node v in the network is given two pieces of information: the identity of the root node r, and the distance between v and r in T. Now it is fairly easy to see that this information suffices to turn T into a locally checkable spanning tree. For example, if T contained a cycle, at least one node would notice it, as the distance labels would not be consistent with the structure of T, and if T consisted of two components, some pair of adjacent nodes would notice that they do not agree on the identity of the root node. No matter how we assign the proof labels, at least one node will immediately detect if T is not a spanning tree.

Locally checkable proofs are very strong in the sense that they tolerate arbitrary failures and even malicious tampering: if all nodes accept the proof, we can be sure that the routing tables are globally consistent.

Obviously, the construction of a spanning tree is not a local task, and neither is the construction of a locally checkable spanning tree. However, the key point is that proof labeling schemes make the routing task of *verification* lightweight, and hence we can trigger a more expensive task of *updating* routing tables in a timely manner in precisely those situations in which it is needed.

4. FROM SDN TO LOCAL ALGORITHMS

So far, we have shown how results in the field of locality-sensitive computing can provide guidelines on how to design, manage, and verify an SDN network. We will now argue that the benefit of this comparison is bidirectional: The SDN network model comes with an interesting twist that generalizes the problems typically studied in the context of local algorithms. In particular, we will introduce the so-called *supported locality model* which opens a wide range of new theoretical problems.

Informally, we can identify *two different hurdles* that often prevent us from solving problems with local algorithms: (1) challenges related to symmetry breaking, and (2) challenges related to finding good solutions to optimization problems. One of the hurdles disappears in the context of SDN.

To illustrate our point, let us consider the example of computing *matchings*; a similar picture emerges with many other classical graph problems [19]. Inspired by the link assignment task (Example 1), let us study the following closely related problems:

(M1) maximal matching
(M2) maximal matching in a bicolored graph
(M3) maximum matching
(M4) maximum matching in a bicolored graph
(M5) fractional maximum matching

(Here a *bicolored graph* is a bipartite graph that is colored with two colors. This is precisely what we had in the link assignment task; one color class consisted of customer sites and the other color class consisted of the points-of-presence. A *fractional matching* is the usual linear programming relaxation of the maximum matching problem; it is a packing linear program.)

We can classify the above graph problems as follows. (For simplicity, we will focus on the case of bounded-degree graphs: all nodes have degree at most $\Delta = O(1)$.)

Optimization:

- (M1), (M2): we only need to find a *feasible* solution.
- (M3), (M4), (M5): we ask for an *optimal* solution.

Symmetry breaking:

- (M1), (M3): symmetry-breaking is *required.* For example, if our input graph is a symmetric cycle, some but not all edges have to be chosen.
- (M2), (M4): symmetry-breaking is required, but we are already *given* a two-coloring of the input graph, which breaks symmetry.
- (M5): symmetry-breaking is not needed; the problem is *trivial* in a symmetric graph.

Locality:

- (M1), (M3): cannot be solved with a local algorithm—symmetry breaking is impossible [16].
- (M2): easy to solve with a local algorithm [7].
- (M4), (M5): cannot be solved *optimally* with a local algorithm—these are inherently global problems.
- (M4), (M5): can be *approximated* arbitrarily well with local algorithms [1, 14].

All of this applies to deterministic local algorithms and the classical models of distributed computing, most notably the LOCAL and CONGEST models [18].

In what follows, we will argue that *the standard models of distributed computing are overly pessimistic.* More specifically, we make the following claims:

- ***Symmetry-breaking** issues need not be a hurdle in the design of efficient protocols for SDN networks.*
- *However, prior results related to the difficulty of finding good solutions to **optimization** problems still apply.*

To see why this is the case, note that we can identify two very different time scales in SDN networks: (1) Constructing the network, e.g., the physical deployment of new controllers and the creation of new controller domains. (2) Reacting to new events, e.g., finding optimal routes based on the current traffic patterns. We can exploit this in the design of protocols:

(1) We can *break symmetry* already while we are constructing the network, and we can use *non-local protocols* to do that. As a simple example, every time we deploy a new controller, we could trigger a global recomputation that gathers information related to the *physical* network topology.

(2) However, we need to know the current *logical* state of the system (e.g., current traffic) before we can *produce the output.* As we want to respond to events quickly, we would like to use a *local* protocol.

An appropriate model of computation needs to take these opportunities into account.

We suggest the following two-stage model of computation. We have an underlying network G that represents the physical network topology. The current logical state of the network is represented as a *subgraph* H of G; here graph H may be annotated with additional information related to the state of the network (e.g., forwarding set, Network Information Base, etc.). Each node v of network G is given the following information:

(1) G: full *topological* information about G.

(2) $H[v, r]$: full *state information* about radius-r neighborhood of v in H.

Based on this information (and nothing else), node v has to produce its own local output. The local outputs of the nodes must form a globally consistent solution to the task at hand. A crucial aspect is that we are studying problems related to the structure of graph H—for example, we would like to find a maximal matching in graph H. However, we have additional knowledge of the underlying network G, which lets us get rid of most issues related to symmetry breaking.

We will refer to this model of local computing with additional state information on H as the *supported model*. Similarly to the classical LOCAL and CONGEST models, we can have the variants of *supported*-LOCAL (no restriction on message size) and *supported*-CONGEST (with message size restrictions) model.

As a simple example, it is easy to show that problem (M1) admits a local algorithm in the supported model. Another example of the supported models is related to the classical dominating set problem. The dominating set algorithm by Friedman and Kogan [5] consists of two phases: symmetry breaking (distance-2 coloring) and optimization (greedy). In the supported model we can solve both phases with a local algorithm—in essence, we can pre-compute a distance-2 coloring of the underlying network G, which provides an appropriate coloring of the subgraph H as well.

There are many other straightforward examples of the power of the supported models, but also many open problems. In particular, many classical lower bound results no longer apply, which makes it more challenging to understand the fundamental limitations of local algorithms in the supported model.

5. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful feedback. This work was supported in part by the Academy of Finland, Grant 252018, and the EU projects OFELIA and CHANGE.

6. REFERENCES

[1] Matti Åstrand, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. Local algorithms in (weakly) coloured graphs, 2010. Manuscript, arXiv:1002.0125 [cs.DC].

[2] Andrzej Czygrinow, Michał Hańćkowiak, Edyta Szymańska, and Wojciech Wawrzyniak. Distributed 2-approximation algorithm for the semi-matching problem. In *Proc. 26th Symposium on Distributed Computing (DISC 2012)*, volume 7611 of *LNCS*, pages 210–222, Berlin, 2012. Springer.

[3] Patrik Floréen, Petteri Kaski, Valentin Polishchuk, and Jukka Suomela. Almost stable matchings by truncating the Gale–Shapley algorithm. *Algorithmica*, 58(1):102–118, 2010.

[4] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *Proc. 16th International Conference on Functional Programming (ICFP 2011)*, pages 279–291, New York, 2011. ACM Press.

[5] Roy Friedman and Alex Kogan. Deterministic dominating set construction in networks with bounded degree. In *Proc. 8th International Conference on Distributed Computing and Networking (ICDCN 2011)*, volume 6522 of *LNCS*, pages 65–76, Berlin, 2011. Springer.

[6] Mika Göös and Jukka Suomela. Locally checkable proofs. In *Proc. 30th Symposium on Principles of Distributed Computing (PODC 2011)*, pages 159–168, New York, 2011. ACM Press.

[7] Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *Proc. 9th Symposium on Discrete Algorithms (SODA 1998)*, pages 219–225, Philadelphia, 1998. SIAM.

[8] Nicholas J. A. Harvey, Richard E. Ladner, László Lovász, and Tami Tamir. Semi-matchings for bipartite graphs and load balancing. *Journal of Algorithms*, 59(1):53–78, 2006.

[9] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proc. 1st Workshop on Hot Topics in Software Defined Networking (HotSDN 2012)*, pages 19–24, New York, 2012. ACM Press.

[10] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proc. 1st Workshop on Hot Topics in Software Defined Networking (HotSDN 2012)*, pages 7–12, New York, 2012. ACM Press.

[11] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*, pages 351–364, Berkeley, 2010. USENIX Association.

[12] Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007.

[13] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.

[14] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *Proc. 17th Symposium on Discrete Algorithms (SODA 2006)*, pages 980–989, New York, 2006. ACM Press.

[15] Paul Lappas. SDN use case: Multipath TCP at Caltech and CERN. Project Floodlight Blog, December 2012.

[16] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

[17] Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.

[18] David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 2000.

[19] Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys*, 45(2):24:1–40, 2013.