Detour Planning for Fast and Reliable Failure Recovery in SDN with OpenState

Antonio Capone*, Carmelo Cascone*†, Alessandro Q.T. Nguyen*, Brunilde Sansò†
* Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy
Email: antonio.capone@polimi.it, alessandro.nguyen@mail.polimi.it
† Département de génie électrique, École Polytechnique de Montréal, Canada
Email: carmelo.cascone@polymtl.ca, brunilde.sanso@polymtl.ca

***Abstract*—A reliable and scalable mechanism to provide protection against a link or node failure has additional requirements in the context of SDN and OpenFlow. Not only it has to minimize the load on the controller, but it must be able to react even when the controller is unreachable. In this paper we present a protection scheme based on precomputed backup paths and inspired by MPLS "crankback" routing, that guarantees instantaneous recovery times and aims at zero packet-loss after failure detection, regardless of controller reachability, even when OpenFlow's "fast-failover" feature cannot be used. The proposed mechanism is based on OpenState, an OpenFlow extension that allows a programmer to specify how forwarding rules should autonomously adapt in a stateful fashion, reducing the need to rely on remote controllers. We present the scheme as well as two different formulations for the computation of backup paths.**

I. Introduction

Failure management is one of the fundamental instruments that allows network operators to provide communication services that are much more reliable than the individual network components (nodes and links). It allows reacting to failures of network components by reconfiguring the resource allocation so as to make use of the surviving network infrastructure able to provide services.

Traditionally, failure resilience has been incorporated in distributed protocols at the transport (like e.g. SDH) and/or network layer (like e.g. MPLS) with some optimization of resources pre-computed for a class of possible failures (like e.g. single link or node failures) and implemented with signaling mechanisms used to notify failures and activate backup resources.

With the introduction of the revolutionary and successful paradigm of Software-defined Networking (SDN), the traditional distributed networking approach is replaced with a centralized network controller able to orchestrate traffic management through the programing of low-level forwarding policies into network nodes (switches) according to simple abstractions of the switching function like that defined in OpenFlow with the match/action flow table [1].

Even if SDN and OpenFlow provide huge flexibility and a powerful platform for programming any type of innovative network application without the strong constraints of distributed protocols, they can make the implementation of important traditional functions, like failure resilience, neither easy nor efficient, since reaction to events in the network must always involve the central controller (notification of an event and installation of new forwarding rules) with non-negligible delays and signaling overheads.

978-1-4799-7795-6/15/$31.00 ©2015 IEEE

New versions of OpenFlow [2] have recently introduced a mechanism, namely fast-failover, for allowing quick and local reaction to failures without the need to resort on central controller. This is obtained by instantiating multiple action buckets for the same flow entry, and applying them according to the status of links (active or failed). However, fast-failover can be used only to define local detour mechanisms when alternative paths are available from the node that detects the failure. Depending on network topology and the specific failure, local detour paths may not be available or they may be inefficient from the resource allocation point of view.

A recent proposal (by some of the authors) [3], [4], named OpenState, has extended the data plane abstraction of OpenFlow to include the possibility for switches to apply different match-actions rules depending on states and to make states evolve according to state machines where transitions are triggered by packet-level events.

In this paper, we propose a new approach to failure management in SDN which exploits OpenState ability to react to packet-level events in order to define a fast path restoration mechanism that allows to reallocate flows affected by failure by enabling detours in any convenient nodes along the primary path. No specific signaling procedure is adopted for triggering detours, rather the same packets of the data traffic flows are tagged and forwarded back to notify nodes of the failure and to induce a state transition for the activation of pre-computed detours.

We define optimization models for the computation of backup paths for all possible single node and link failures that consider multiple objectives including link congestion level, distance of the reroute point from the failure detection point, and level of sharing of backup paths by different flows. We show that the MILP (Mixed Integer Linear Programming) formulations proposed are flexible enough to incorporate the optimization of the OpenFlow fast-failover reroutes as a special case and that path computation for all possible failure scenarios can be performed within reasonable time for realistic size networks with state-of-the-art solvers (cplex).

The reminder of the paper is presented as follows. In Section II we first present an overview of OpenState and next we present the proposed failure recovery scheme in Section III. Related work is reviewed in Section IV and in Section V two

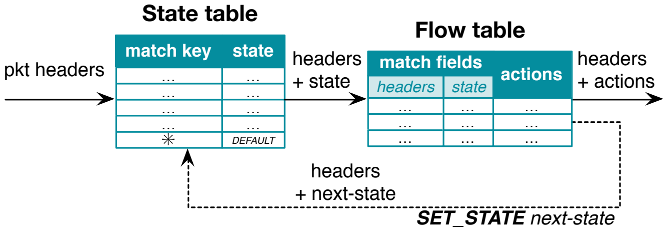


Fig. 1: Simplified packet flow in OpenState.

modelling formulations are presented. Computational results are discussed in Section VI. Conclusions are provided in Section VII.

II. OpenState

The most prominent instance of SDN is OpenFlow, which, by design, focuses on an extreme centralization of the network intelligence at the controller governing switches, which in turn are considered dumb. In OpenFlow, adaptation and reconfiguration of forwarding policies can only be performed by remote controllers, with a clear consequence in terms of overhead and control latency. OpenState is an OpenFlow extension that enables mechanisms for controllers to offload some of their control logic to switches. In OpenState, the programmer is able to define forwarding rules that can autonomously adapt in a stateful fashion on the basis of packet-level events. The motivation beside OpenState is that control tasks that require only switch-local knowledge are unnecessarily handled at the controller, and thus can be offloaded to switches, while maintaining centralized control for those tasks that require global knowledge of the network.

OpenState has been designed as an extension (superset) of OpenFlow. In OpenState the usual OpenFlow match/action flow table is preceded by a state table that contains the so called “flow-states”. First, packets are matched against the state table using only a portion of the packet header (a programmable lookup-key), a state lookup operation is performed and a state label (similar to OpenFlow’s metadata) is appended to packet headers. A DEFAULT state is returned if no row is matched in the state table. Packets are then sent to the flow-table where the usual OpenFlow processing is performed, while a new SET_STATE action is available to insert or rewrite rows of the state table with arbitrary values. Figure 1 illustrates the packet flow in the two tables. OpenState allows also to match packets using “global-states”, so called because, in contrast to flow-states, these are globally valid for the whole switch (datapath) and not just for a given flow. By using flow-states and global-states a programmer can define flow entries that apply to different scenarios, and by using state transition primitives she can control how those scenarios should evolve.

OpenState has been showed to bring tangible benefits in the implementation of fundamental network applications [4]. An open-source implementation of an OpenState controller and switch can be found at [5], along with a modified version of Mininet and few application examples.

III. Proposed Approach

The approach we take is similar to that used in crankback signaling [6]. In the context of end-to-end QoS in MPLS and GMPLS with RVSP-TE, when a connection or flow setup fails because of a blocked link or node, crankback is a signaling technique in which a notification of the failure is backtracked along the flow path, from the upstream node that faces the blockage up to the first node (called “repair point”) that can determine an alternative path to the destination node.

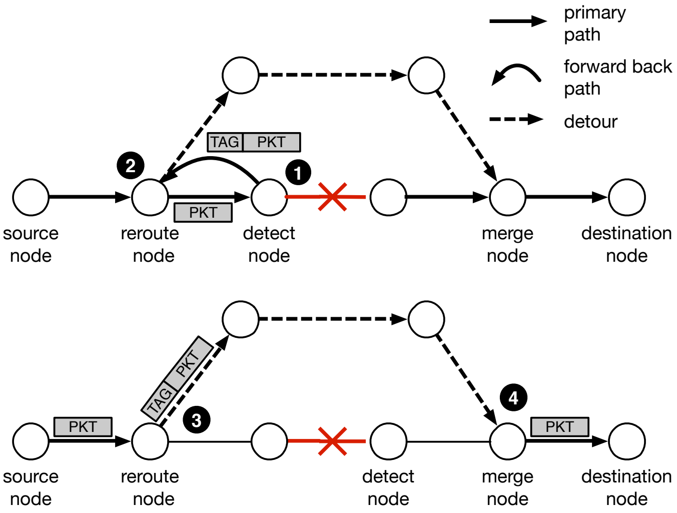


Fig. 2: Example of failure recovery with OpenState: in (1) the upstream node detects the failure, tags the packet and forwards it back. In (2) the reroute node receive the tagged packet, executes a state transition and forward the packet on the detour. In (3) all the packets received for the considered demand after the state transition, will be tagged and forwarded on the detour. Finally in (4), at the end of the detour, the tag is popped and the packet is forwarded on the primary path, towards its destination node.

Our solution is based on the same idea, but with the major difference that, upon link or node failure, the same data packets, and not a notification, can be sent back on their original path. We distinguish two situations: (i) the node which detects the failure is able to reroute the demand, and (ii) the packet must be forwarded back on it’s primary path until a convenient reroute node is encountered. In the first case, solutions like OpenFlow’s fast-failover already guarantee almost instantaneous protection switching without controller intervention, while in the second case it would be impracticable to signal the failure to other nodes without the intervention of the controller. The novelty of our approach is given by the fact that, in the second case, a crankback approach is performed using the same data packets, which are first tagged (e.g. with a MPLS label containing information on the failure event) and then sent back trough the primary path. A reroute node who receives the tagged packet will be able to respond to the failure by rerouting the tagged packets and by enabling a detour for all subsequent packets. That said, only the first packets of the flow are sent back from the detect node. As soon as the first tagged packet is processed by the reroute node, a state transition is performed in the OpenState switch, and all subsequent packets coming from the source node will be forwarded on the detour. An example of the mechanism described so far is summarized in Figure 2.

With this approach, flow-states are used to distinguish the forwarding of each traffic demand at each switch. The DEFAULT state implies that the demand can be forwarded towards the next node on the primary path, other arbitrary states are used to describe the specific failure that can affect the demand, so that the same reroute nodes can react differently according to the specific failure event. Global-states are instead used to describe the operational status of switch ports (up or down). In this case global-states are completely equivalent to "port liveness" states used by OpenFlow fast-failover feature.

Our proposal is currently independent of the way failures are detected, because it does not influence the modeling aspect of the solution. We assume it could be implemented either via Loss Of Signal (LOS) or Bidirectional Forwarding Detection (BFD) [7] mechanisms. In both cases, as soon as the state of the failed port is updated, our solution guarantees instantaneous reaction with ideally zero packet-loss.

IV. Related Work

Failure management in SDN is a topic that has been already explored by the research community. In [8] the authors analyze the case of restoration for OpenFlow networks, showing how hard it is to achieve fast (< 50ms) recovery times in large networks. Restoration is also taken into consideration in [9], where the controller is entitled to monitor link status on the network, and, in case of failure, it computes a new path for the affected demand and replaces or deletes flow entries in switches, accordingly. In [10] an end-to-end path protection scheme is proposed: OpenFlow 1.1 is extended by implementing in the switches a monitoring function that allows to reduce processing load on the controller. Such a function is used in conjunction with OpenFlow fast-failover feature, thus allowing nodes to autonomously react to failures by switching to a precomputed end-to-end backup path. In [11] a segment protection mechanisms is proposed only for the case of link failure. Backup paths are pre-installed, and OpenFlow is extended to enable switches to locally react to connected failed links. Another way to reduce the load at the controller is presented in [12]. The authors propose a centralized monitoring scheme and a model to reduce the number of monitoring iterations that the controller must perform in order to check all links. A completely different and creative approach is proposed in [13], where classic graph search algorithms are presented to implement a solution based on the OpenFlow fast-failover scheme, where backup paths are not known in advance but nodes implement an algorithm to randomly try new routes to reach the destination.

V. Problem Formulation

Let $G(N, A)$ be a symmetric directed graph so that N represents the set of network switches, and A the set of links between switches. The demands are assumed to be known in advance. Also assumed is the fact that each demand will be routed using a primary path optimized as a shortest-path with link capacity constraints. Our main problem then focuses on the evaluation of backup paths for each demand, for every possible single failure scenario in the primary path. The significance of a failure scenario will be clearly indicated in the next subsection. For comparison purposes we also present, at the end of the Section, a congestion avoidance version of the same backup path problem.

A. Backup Path Problem Formulation

In the forthcoming model, we refer to "failure detection event" rather than simply "failure state" to indicate that a failure has been perceived. Moreover, instead of making an a priori distinction between the case of link and the case of node failure, a "fault detection event" $f = (n, m)$ may be either. The notation simply indicates that node n detects a failure while transmitting to a downstream node m. Therefore two distinct situations are considered: (i) a failure on link (n, m) (e.g. disconnected or truncated cable, etc.) and (ii) a scenario where downstream node m fails implying the disconnection of all its adjacent links. When evaluating the backup path for a given demand, we always consider the worst-case scenario of a node failure, thus completely avoiding to forward packets to m, except for the case where m is also the destination node of the considered demand ($m = t_d$). In such a case, we try to deliver packets to m avoiding the failed link (n, m).

Let us now define the following parameters:

Parameters

D set of demands to be routed;

s_d source node of demand d;

t_d destination node of demand d;

β_{dij} is equal to 0 if link (i, j) belongs to the primary path for demand d, otherwise 1;

b_d requested bandwidth for demand d;

c_{ij} total capacity of link (i, j);

w_{cap} percentage of the link capacity available;

F set including all the possible failure detection events (n, m) that can affect at least one primary path;

D^{nm} subset of D including all the demands affected by the failure detection events (n, m);

D_1^{nm} subset of D^{nm} including all the demands d affected by the failure detection event (n, m), when m is not the destination node of the considered demand and thus $m \neq t_d$;

D_2^{nm} subset of D^{nm} including all the demands d affected by the failure detection event (n, m), where m is the destination node of the considered demand and thus $m = t_d$;

L^m subset of A that will include all the links incident to node m;

u_{ij}^{nm} represents the used capacity of link (i, j) when link (n, m) fails. Note that in this parameter we consider only the link capacity allocated for those demands for which the primary path does not include neither (m, n) or (n, m);

v_{ij}^{m} is the used capacity of link (i, j) in case of failure for node m. In this case we consider only the link capacity allocated for those demands that are not affected by a failure of node m, in other words those demands which primary path does not include any of the links incident to m;

p_d^k represents the link (i, j) in the k-th position of the primary path for demand d, where $k = 1$ is intended as the first link of the primary path starting from node s_d;

λ_d^{nm} is the number of nodes that a packet of demand d traverses on the primary path, before reaching node n of failure detection event (n, m). $\lambda_d^{nm} = 0$ means that the failure has been detected by the first node of the path.

Decision variables

y_{dij}^{nm} is equal to 1 if link (i,j) belongs to the backup path of demand d in case of failure detection event (n,m), otherwise 0;

h_d^{nm} non-negative integer that represents the number of backward hops that a tagged packet of demand d must perform in case of failure detection event (n,m), before reaching the reroute node that will enable the detour. When $h_d^{nm}=0$ we mean that node n that detected the failure is also the reroute node;

z_{dij} equal to 0 if (i,j) is not used by any backup path (for every possible failure) for demand d, otherwise 1.

Objective Function

$$\begin{aligned} min \quad & \sum_{(n,m)\in F}\sum_{d\in D^{nm}} w_h h_d^{nm} \\ & + \sum_{(n,m)\in F}\sum_{d\in D^{nm}}\sum_{(i,j)\in A} w_y y_{dij}^{nm} \\ & + \sum_{d\in D}\sum_{(i,j)\in A} w_z \beta_{dij} z_{dij} \end{aligned} \tag{1}$$

The objective function is composed of three weighted terms. The first minimizes the length of the reverse path that tagged data packets must travel in case of failure. The second minimizes the length of backup paths. The third term minimizes the number of links allocated to the backup paths for a given demand, in other words we want more backup paths of the same demand to share the same links. By using the three weights w_h, w_y, and w_z we are able to characterize the behavior of the objective function in different ways.

Link availability constraints

$$\sum_{(i,j)\in L^m} y_{dij}^{nm} \le 0 \qquad \forall (n,m)\in F, \forall d \in D_1^{nm} \tag{2}$$

$$y_{dnm}^{nm} + y_{dmn}^{nm} \le 0 \qquad \forall (n,m)\in F, \forall d \in D_2^{nm} \tag{3}$$

These constraints disable the use of certain links when evaluating the backup path for a given demand.

Link capacity constraints

$$u_{ij}^{nm} + \sum_{d\in D^{nm}} b_d y_{dij}^{nm} + \sum_{e\in D^{mn}} b_e y_{eij}^{mn} \le w_{cap} c_{ij} \qquad \forall (n,m)\in F, \forall (i,j)\in L \tag{4}$$

$$v_{ij}^{m} + \sum_{\substack{n\in N:\\ (n,m)\in F}} \sum_{d\in D^{nm}} b_d y_{dij}^{nm} \le w_{cap} c_{ij} \qquad \forall m \in N, \forall (i,j)\in L \tag{5}$$

The above constraints insure that for every possible failure, when allocating the backup paths, the link capacity must be respected. The first set of constraints is specific for the case of link failure, while the second set is specific for the case of node failure. Because we do not know the exact nature of a failure detection event, we want our solution to be valid (in terms of resource allocation) in case of both link and node failure.

Flow conservation constraints

$$\sum_{\substack{j\in N:\\ (i,j)\in A}} y_{dij}^{nm} - \sum_{\substack{j\in N:\\ (j,i)\in A}} y_{dji}^{nm} = \begin{cases} 1, & \text{if } i = s_d; \\ -1, & \text{if } i = t_d; \\ 0, & \text{otherwise.} \end{cases} \qquad \forall i \in N, \forall (n,m)\in F, \forall d \in D^{nm} \tag{6}$$

These constraints assure that there is continuity in backup paths.

Cycle avoidance constraints

$$\sum_{\substack{j\in N:\\ (i,j)\in L}} y_{dij}^{nm} \le 1 \qquad \forall i \in N, \forall (n,m)\in F, \forall d \in D^{nm} \tag{7}$$

These constraints avoid the creation of cycles in the backup paths.

Reverse path constraints

$$\sum_{\substack{k=1:\\ (i,j)=p_d^k}}^{\lambda_d^{nm}} (1 - y_{dij}^{nm}) \le h_d^{nm} \qquad \forall (n,m)\in F, \forall d \in D^{nm} : \lambda_d^{nm} \ne 0 \tag{8}$$

These constraints are needed to evaluate the variable h_d^{nm}.

Capacity use constraints

$$z_{dij} \ge y_{dij}^{nm} \qquad \forall (i,j)\in A, \forall (n,m)\in F, \forall d \in D^{nm} \tag{9}$$

These constraints are needed to evaluate the variable z_{dij}.

Having reviewed the main backup path formulation, we now present, in the the next subsection a congestion avoidance formulation to be used for comparison purposes.

B. Congestion Avoidance Formulation

Let us first define the following additional variables:

μ_{ij} represents the maximum capacity used on link (i,j) w.r.t. all possible failure detection events;

ϕ_{ij} represents the cost of using link (i,j) when the capacity used is μ_{ij}.

The problem can then be formulated as follows

Objective function

$$min \sum_{(i,j)\in A} \phi_{ij} \tag{10}$$

This new objective function is a classical non-linear congestion related optimization function that aims at minimizing the load on each link. As we will later see, the function will be linearized in order to treat the integer problem.

TABLE I: Topologies summary

Topology	$\mid N \mid$	$\mid A \mid$	$\mid N_{edge} \mid$	$\mid N_{core} \mid$	$\mid D \mid$
Polska	12	36	9	3	72
Norway	27	102	16	11	240
Fat tree	20	64	8	12	56

Link capacity constraints

Previous constraints (2), (3) and (6) are maintained, while link capacity constrains (4) and (5) are substituted by the following:

$$u_{ij}^{nm} + \sum_{d\in D^{nm}} b_d y_{dij}^{nm} + \sum_{e\in D^{mn}} b_e y_{eij}^{mn} \le \mu_{ij}$$
$$\forall (n,m) \in F, \forall (i,j) \in L \qquad (11)$$

$$v_{ij}^{m} + \sum_{\substack{n\in N:\\ (n,m)\in F}} \sum_{d\in D^{nm}} b_d y_{dij}^{nm} \le \mu_{ij}$$
$$\forall m \in N, \forall (i,j) \in L \qquad (12)$$

$$\mu_{ij} \le w_{cap} c_{ij} \qquad \forall (i,j) \in L \qquad (13)$$

(11) and (12) evaluate the maximum load on link (i,j) for all considered failure detection events (m,n), while (13) stipulates that even for the maximum value the capacity of the link must be respected.

Linearization constraints

Given that ϕ_{ij} in (10) is a non-linear performance function, it should be linearized by the following constraints:

$$\phi_{ij} \ge \frac{\mu_{ij}}{w_{cap}c_{ij}} \qquad \forall (i,j) \in A \qquad (14)$$

$$\phi_{ij} \ge 3\frac{\mu_{ij}}{w_{cap}c_{ij}} - \frac{2}{3} \qquad \forall (i,j) \in A \qquad (15)$$

$$\phi_{ij} \ge 10\frac{\mu_{ij}}{w_{cap}c_{ij}} - \frac{16}{3} \qquad \forall (i,j) \in A \qquad (16)$$

$$\phi_{ij} \ge 70\frac{\mu_{ij}}{w_{cap}c_{ij}} - \frac{178}{3} \qquad \forall (i,j) \in A \qquad (17)$$

$$\phi_{ij} \ge 500\frac{\mu_{ij}}{w_{cap}c_{ij}} - \frac{1468}{3} \qquad \forall (i,j) \in A \qquad (18)$$

This set of equations represent the linearized load cost function shown in Fig. 3.

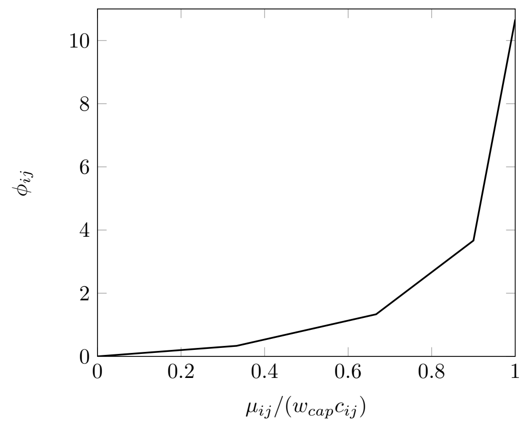


Fig. 3: Load cost function ϕ_{ij}

VI. Computational Results

The model was tested on three different network topologies portrayed in Figure 4. Two real backbone topologies, namely Polska and Norway, taken from [14], and a fat tree, which is an example of a symmetric topology well known for its degree of fault-resiliency [15], and widely used in data centers. For each topology, nodes are divided in two sets: edge nodes and core nodes. Edge nodes act as source and destination of traffic while core nodes are only in charge of routing.

As mentioned in Section V, one of the inputs of the model is a set of primary paths evaluated as shortest paths for every traffic demand. Once such input was known, backup paths were found by varying weights w_h,w_y, and w_z of objective function (1). Three types of instances were evaluated for comparison purposes: those referring to the backup problem with a given set of weights, those referring to the congestion avoidance formulation and those referring to a classic end-to-end path protection formulation. A summary of such instances is given below:

BP$_{111}$ all three terms of the objective function are taken into account;

BP$_{100}$ only the first term is considered, thus the model is forced to find a solution that minimizes the length of the reverse path, converging to a solution where failure detection node and reroute node are the same;

BP$_{010}$ only the second term is considered by minimizing the length of backup paths from s_d to t_d;

BP$_{001}$ only the third term is considered, thus trying to minimize the number of links allocated for all backup paths of each demand;

BP$_{\text{CA}}$ congestion avoidance formulation of the BP problem, minimizing the maximum load for each link;

E2E classic end-to-end path protection problem formulation.

The instances were executed assuming 2 different link capacity sets $c_{i,j}$: (i) capacity is set to the minimum value to obtain a feasible solution, and (ii) links are over-provisioned with very high capacity. For each test the requested bandwidth for each demand is always set to $b_d = 1$, and the available link capacity parameter is fixed to $w_{cap} = 80\%$.

The models were formalized and solved to optimality with AMPL-Cplex, using PCs with 8 CPU cores Intel Core i7 and 8GB of RAM. For all executions a solution was found in less than 30 seconds, except for the case of BP$_{\text{CA}}$ evaluated for the Norway topology, where the execution required about ten minutes.

The solutions were compared evaluating the trade-off with respect to the following parameters:

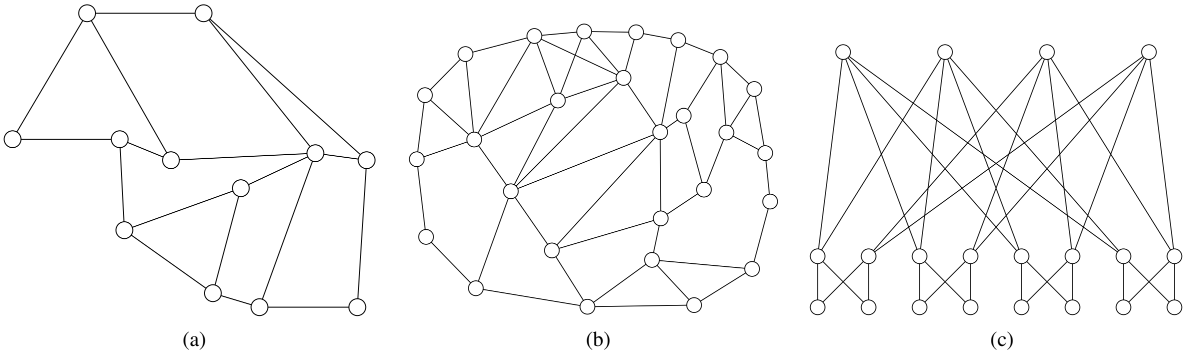


Fig. 4: Network topologies used in test instances: (a) Polska, (b) Norway, and (c) Fat tree

- **Backup path length:** this measure was assessed with respect to the primary path length. A value of 100% means that the length of the backup path is twice the primary path length, whereas 0% indicates that the two paths have the same length.
- **Link capacity occupation:** is the percentage of the total link capacity allocated for all primary and backup paths that use the considered link.
- **Reverse path length:** is the portion of the primary path that a tagged packet has to traverse before being rerouted. A value of 100% indicates that the packet has to go back to the source node of the demand, while a 0% means that the packet is rerouted from the same node that detected the failure.

The complete set of results is shown in Table II and in chart form in Figure 5.

In all instances BP_{111} offers the best trade-off in terms of backup path length and reverse path length, with no major drawbacks. $\mathrm{BP}_{\mathrm{CA}}$ produces better solutions in terms of link capacity occupation, especially when considering instances with minimum capacity c_{ij} (see Figures 5c, 5f and 5i for a clearer view). The drawback of using $\mathrm{BP}_{\mathrm{CA}}$ is represented by longer backup paths. In fact, for Norway and Polska topologies, $\mathrm{BP}_{\mathrm{CA}}$ produces solutions with the longest backup paths, about the double in both cases (Figures 5a and 5d). However, note that in the case of an on-line scenario $\mathrm{BP}_{\mathrm{CA}}$ would guarantee more residual capacity and thus a higher probability of accepting new traffic demands.

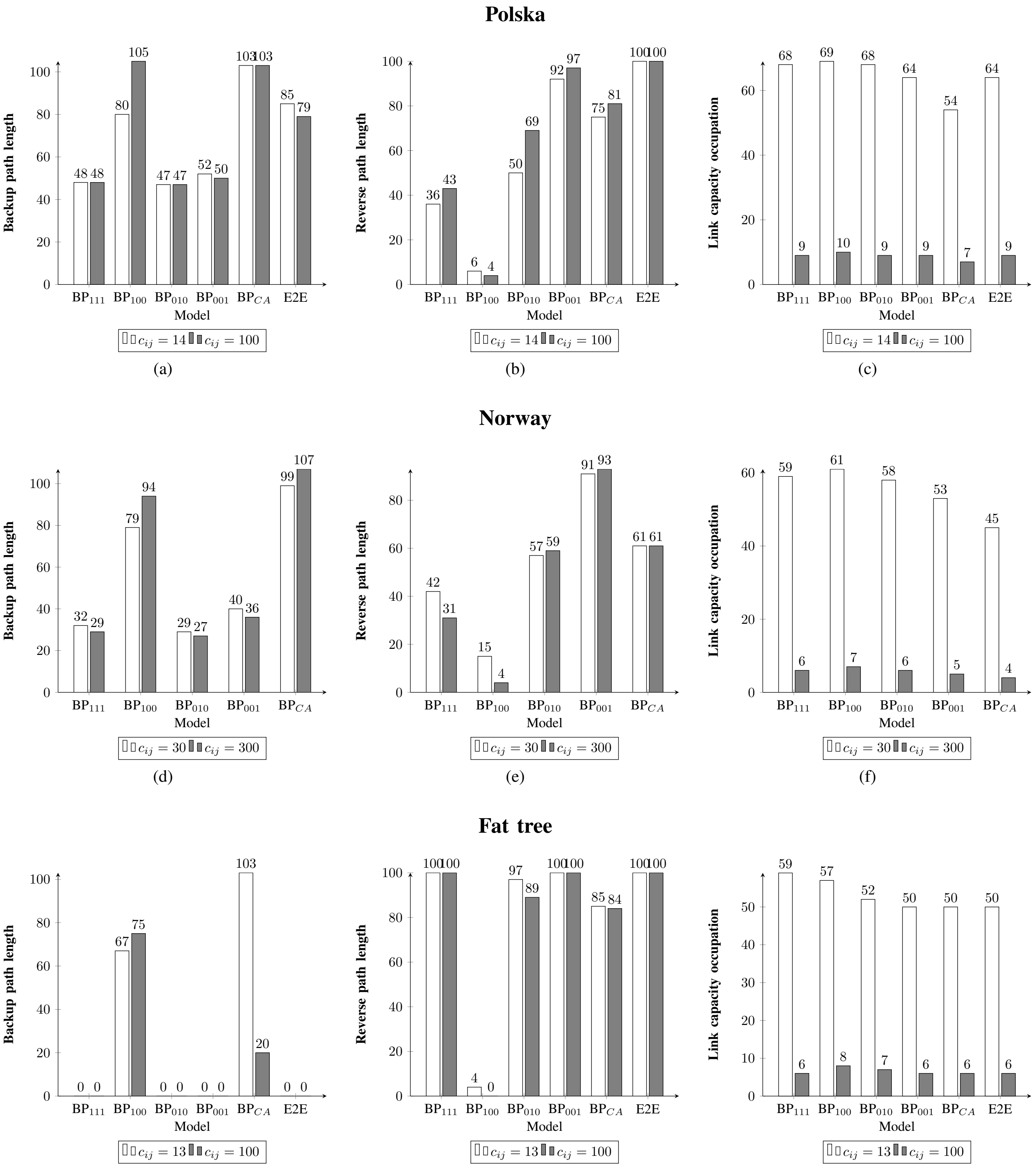
Concerning the reverse path length, the best solution is obtained with configuration BP_{100} (Figures 5b, 5e, 5h). The drawback in this case is represented by longer backup paths, about the double when compared to primary paths. It is interesting to note that for the fat tree topology with $c_{ij} = 100$ (Figure 5h) BP_{100} returns a solution with reverse path length equal to 0%. This is worth mentioning because this solution would be suitable to be implemented with OpenFlow fast-failover, where detect node and reroute node are always the same. Unfortunately such a solution is not always feasible, as it strongly depends on topology and capacity constraints. Indeed, for all other cases, BP_{100} is unable to provide a solution with 0% reverse path length. Thanks to this result we can show how our solution based on OpenState, which is able to handle reverse paths, guarantees an higher degree of fault-resiliency when compared to a solution based on OpenFlow fast-failover.

It is also interesting to note that for the Norway topology the given set of primary paths has no feasible solution for the E2E model. This is due to the fact that in the classic formulation of E2E path protection, primary paths and backup paths must be evaluated at the same time, thus avoiding the situation where for a given primary path is impossible to find a completely disjoint backup path. We show in this case the flexibility of our approach by always providing a feasible solution.

Finally, it is interesting to note how for the case of the fat tree topology, the results obtained from BP_{111} are the same of the E2E model, always having backup path length equal to primary paths, and reverse path length equal to 100%. This means that in case of failure, packets will be always rerouted from the source node of the demand. In this case a solution adopting OpenState would guarantee less disruption thanks to the fact that nodes would be able to automatically switch to the backup path, whereas OpenFlow would require to forward packet to the controller to enable the backup path at the source node by installing the respective forwarding rules.

VII. Conclusion

In this paper we have presented a new failure management framework for SDN and a mathematical modeling approach specifically designed to exploit the capabilities of OpenState. The framework considers both single link and single node failure. The protection scheme is based on the idea that, upon failure detection, packets can be tagged and backtracked along the primary path to signal the failure to the first convenient reroute node, automatically establishing a detour path. Such scheme aims at having zero packet loss after failure detection, and doesn't require controller intervention. The models were tested on three well-known topologies and comparative results were obtained, showing the superiority of the scheme with respect to a classic end-to-end path protection scheme an with respect to an approach based on the OpenFlow fast-failover mechanism. We are currently working on the dimensioning problem and developing the OpenState application to experimentally validate the proposed solution.



(a) (b) (c) (d) (e) (f) (g) (h) (i)

Fig. 5: Result charts for the three topology examinated

TABLE II: Computational results

Instance	Model	Backup path length			Link capacity occupation			Reverse path length		
		min	*max*	*avg (var)*	*min*	*max*	*avg (var)*	***min***	*max*	*avg (var)*
Polska $c_{ij} = 14, \forall(i,j) \in A$	BP_{111}	0%	300%	48% (61%)	29%	79%	68% (10%)	0%	100%	36% (41%)
	BP_{100}	0%	900%	80% (103%)	43%	79%	69% (9%)	0%	100%	6% (19%)
	BP_{010}	0%	300%	47% (61%)	43%	79%	68% (9%)	0%	100%	50% (45%)
	BP_{001}	0%	300%	52% (60%)	43%	79%	64% (12%)	0%	100%	92% (24%)
	BP_{CA}	0%	700%	103% (123%)	7%	79%	54% (20%)	0%	100%	75% (43%)
	E2E	0%	300%	85% (75%)	29%	79%	64% (13%)	100%	100%	100% (0%)
Polska $c_{ij} = 100, \forall(i,j) \in A$	BP_{111}	0%	300%	48% (61%)	4%	12%	9% (2%)	0%	100%	43% (45%)
	BP_{100}	0%	600%	105% (118%)	5%	16%	10% (2%)	0%	100%	4% (16%)
	BP_{010}	0%	300%	47% (61%)	6%	12%	9% (1%)	0%	100%	69% (43%)
	BP_{001}	0%	300%	50% (61%)	4%	11%	9% (2%)	0%	100%	97% (16%)
	BP_{CA}	0%	700%	103% (136%)	2%	11%	7% (3%)	0%	100%	81% (39%)
	E2E	0%	300%	79% (77%)	3%	12%	9% (2%)	100%	100%	100% (0%)
Norway $c_{ij} = 30, \forall(i,j) \in A$	BP_{111}	0%	500%	32% (55%)	3%	80%	59% (20%)	0%	100%	42% (43%)
	BP_{100}	0%	900%	79% (98%)	17%	80%	61% (18%)	0%	100%	15% (31%)
	BP_{010}	0%	500%	29% (53%)	7%	80%	58% (20%)	0%	100%	57% (42%)
	BP_{001}	0%	500%	40% (54%)	7%	80%	53% (20%)	0%	100%	91% (25%)
	BP_{CA}	0%	1600%	99% (137%)	0%	80%	45% (25%)	0%	100%	61% (49%)
	E2E	-	-	-	-	-	-	-	-	-
Norway $c_{ij} = 300, \forall(i,j) \in A$	BP_{111}	0%	500%	29% (51%)	0%	12%	6% (3%)	0%	100%	31% (39%)
	BP_{100}	0%	1400%	94% (131%)	1%	14%	7% (3%)	0%	100%	4% (17%)
	BP_{010}	0%	500%	27% (52%)	0%	12%	6% (3%)	0%	100%	59% (42%)
	BP_{001}	0%	500%	36% (53%)	0%	12%	5% (3%)	0%	100%	93% (23%)
	BP_{CA}	0%	1400%	107% (138%)	1%	10%	4% (3%)	0%	100%	61% (49%)
	E2E	-	-	-	-	-	-	-	-	-
Fat tree $c_{ij} = 13, \forall(i,j) \in A$	BP_{111}	0%	0%	0% (0%)	15%	77%	59% (13%)	100%	100%	100% (0%)
	BP_{100}	0%	500%	67% (70%)	31%	77%	57% (13%)	0%	100%	4% (13%)
	BP_{010}	0%	0%	0% (0%)	23%	77%	52% (13%)	0%	100%	97% (18%)
	BP_{001}	0%	0%	0% (0%)	15%	77%	50% (14%)	0%	100%	100% (0%)
	BP_{CA}	0%	150%	103% (128%)	0%	77%	50% (15%)	0%	100%	85% (35%)
	E2E	0%	0%	0% (0%)	15%	77%	50% (15%)	100%	100%	100% (0%)
Fat tree $c_{ij} = 100, \forall(i,j) \in A$	BP_{111}	0%	0%	0% (0%)	1%	11%	6% (2%)	100%	100%	100% (0%)
	BP_{100}	0%	400%	75% (75%)	3%	12%	8% (2%)	0%	0%	0% (0%)
	BP_{010}	0%	0%	0% (0%)	2%	12%	7% (2%)	0%	100%	89% (31%)
	BP_{001}	0%	0%	0% (0%)	0%	12%	6% (2%)	100%	100%	100% (0%)
	BP_{CA}	0%	200%	20% (35%)	1%	11%	6% (2%)	0%	100%	84% (36%)
	E2E	0%	0%	0% (0%)	0%	12%	6% (3%)	100%	100%	100% (0%)

Acknowledgment

This work has been funded by NSERC Discovery Grant and by the European Community BEBA project. Luca Pollini and Davide Sanvito were part of the team that coded the algorithms in an OpenState emulator. We are grateful for their input that allowed us to assess upfront the feasibility of the proposed modeling approaches.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling innovation in campus networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] Open Networking Foundation, “OpenFlow switch specification ver 1.4,” Tech. Rep., Oct. 2013.

[3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, “OpenState: programming platform-independent stateful OpenFlow applications inside the switch,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014.

[4] G. Bianchi, M. Bonola, A. Capone, C. Cascone, and S. Pontarelli, “Towards wire-speed platform-agnostic control of OpenFlow switches,” *arXiv preprint arXiv:1409.0242*, 2014.

[5] “OpenState SDN project home page,” http://www.openstate-sdn.org.

[6] A. Farrel, A. Satyanarayana, A. Iwata, N. Fujita, and G. Ash, “Crankback Signaling Extensions for MPLS and GMPLS RSVP-TE,” RFC 4920 (Proposed Standard), Internet Engineering Task Force, Jul. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4920.txt

[7] D. Katz and D. Ward, “Bidirectional Forwarding Detection (BFD),” RFC 5880 (Proposed Standard), Internet Engineering Task Force, Jun. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5880.txt

[8] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, “Software defined networking: Meeting carrier grade requirements,” in *Local Metropolitan Area Networks (LANMAN), 2011 18th IEEE Workshop on*, Oct 2011, pp. 1–6.

[9] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, “Enabling fast failure recovery in OpenFlow networks,” in *Design of Reliable Communication Networks (DRCN), 2011 8th International Workshop on the*, Oct 2011, pp. 164–171.

[10] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Skoldstrom, “Scalable fault management for OpenFlow,” in *Communications (ICC), 2012 IEEE International Conference on*, June 2012, pp. 6606–6610.

[11] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, “OpenFlow-based segment protection in ethernet networks,” *Optical Communications and Networking, IEEE/OSA Journal of*, vol. 5, no. 9, pp. 1066–1075, Sept 2013.

[12] S. Lee, K.-Y. Li, K.-Y. Chan, G.-H. Lai, and Y.-C. Chung, “Path layout planning and software based fast failure detection in survivable OpenFlow networks,” in *Design of Reliable Communication Networks (DRCN), 2014 10th International Conference on the*, April 2014, pp. 1–8.

[13] M. Borokhovich, L. Schiff, and S. Schmid, “Provable data plane connectivity with local fast failover: Introducing Openflow graph algorithms,” in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’14. ACM, 2014, pp. 121–126.

[14] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski, “SNDlib 1.0 survivable network design library,” *Networks*, vol. 55, no. 3, pp. 276–286, 2010.

[15] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, “PortLand: A scalable fault-tolerant layer 2 data center network fabric,” in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, ser. SIGCOMM ’09. ACM, 2009, pp. 39–50.