WiSHFUL

Wireless Software and Hardware platforms for Flexible and Unified radio and network controL

European Commission

Open Call 3

Experiment Report

Pop-Routing On WiSHFUL

POPROW

Call identifier	WiSHFUL-OC3-EXP-EXC
Date of report	31/10/2017
Version number (optional)	1.0
Organisation	DISI: Department of Information Engineering and Computer Science, University of Trento, Italy.
Address	Via Sommarive 9, 38123 Povo (TN), Italy
Coordinator	Prof. Renato Lo Cigno
Authors	Michele Segata, Nicolò Facchi, Leonardo Maccari, Renato Lo Cigno

Table of Contents

Section A. Summary

The goal of POPROW is to test and enhance "Pop-Routing", a technique for wireless mesh link-state routing protocols that tunes the generation frequency of control messages independently for each node of a wireless mesh network as the result of real-time graph analysis performed on the network topology. Pop-Routing is backward-compatible and allows the reduction of the routing tables convergence time without increasing the total amount of control messages. Pop-Routing theory was presented at IEEE Infocom 2016 [1] evaluated via emulation, while a later paper [2] introduced the possibility for centrality computation optimization.

Within the POPROW project, starting from the prototypes, a real implementation has been developed in form of a plug-in for both OLSRv1 and OLSRv2, an external (to the routing daemon) companion process, named Prince, computes the optimal value of the timers and feeds them to the plug-in.

The WiSHFUL infrastructure made it possible to develop the working implementation of Pop-Routing and test it in real conditions, running both functional and performance tests. It is particularly valuable the fact that WiSHFUL provides a setting where scenarios can be completely controlled, and where we can fully control and modify the physical and logical connectivity, so that repeatable, scientific experiments can be run and the engineering process leading to the final implementation is completely sound. We stress that full control of the topology is a key feature since Pop-Routing is dynamic: it changes the frequency with which control messages are generated on a per-node basis depending on the modification of the network topology.

During POPROW we reached four goals:

1. We confirmed that Pop-Routing is able to reduce convergence time in real network scenarios, out of emulation;
2. We stabilized and improved the open source code of Prince, the daemon that implements Pop-Routing, in order to move one crucial step towards the real world implementation of Pop-Routing;
3. We submitted three scientific papers based on POPROW experiments and development;
4. We reached a project maturity that indicate that Pop-Routing can be contributed to standards.

Background and prior research

[1] Leonardo Maccari and Renato Lo Cigno. "Pop-routing: Centrality-based tuning of control messages for faster route convergence." 35th IEEE Int. Conf. on Computer Communications (INFOCOM), 2016.

[2] Leonardo Maccari, Quynh Nguyen, Renato Lo Cigno. "On the Computation of Centrality Metrics for Network Security in Mesh Networks." IEEE Global Communications Conference (GLOBECOM), 2016

Section B. Detailed description

B1 Concept, Objectives, Set-up and Background

B.1.1 Concept & Objectives

Extensive experience with the analysis of large mesh networks shows that there is a threshold beyond which a flat network architecture can not scale when using link-state routing protocols. Today we can push that threshold to hundreds of nodes, but, as long as the total amount of control messages generated by a routing daemon scales linearly with the number of nodes, the only way to raise that threshold is to reduce the frequency of the generation of these messages. This in turn raises the convergence time of the routing tables upon events of topology change, since information is propagated slowly and partly reduces the advantage of using a link-state protocol.

Pop-Routing is a technique that, given the knowledge of the network graph (that is the base information on any link-state protocol), allows each node to automatically and independently tune its own frequency of generation of control messages achieving the global optimal trade-off between the total overhead due to the control messages and the speed of convergence of the routing tables.

So far Pop-Routing was tested only in emulated networks, which confirmed the theoretical analysis done in the paper, but we are aware that emulation hides a lot of the complexity of real world scenarios.

The main objectives of POPROW project are the testing and development of Prince, the open source daemon implementing Pop-Routing on top of the OLSR protocol, in particular:

1. Test the behaviour of Pop-Routing in networks with different features (e.g., size, density, average degree etc.) in presence of perturbations to verify Pop-Routing performance in terms of routing convergence speed;
2. Stabilize Prince to go one step further in its development towards its deployment in real networks.
3. Test new features of Prince like runtime configuration of *HELLO* and *TC* validity multipliers and improved logging.

In the rest of this section we briefly recall the basic concepts required to understand how a generic link-state routing protocol works, with a particular focus on OLSR protocol, and the rationale behind Pop-Routing. This information will be useful for better understanding the experiments set-up and the technical results reported in sections B.1.2 and B.2.

A link-state routing protocol is a protocol in which all the nodes have the knowledge of the full network graph and can populate their routing tables as the result of applying Dijkstra's algorithm to the network graph. OLSR is one of the most studied link-state routing protocols for mesh networks and is characterized by its ability to quickly react to a change in the network topology, for instance, after the failure of a node. This is achieved in two distinct phases: a failure detection phase and a propagation phase. In the detection phase, the failed node's neighbors detect the failure and change their routing table accordingly. In the propagation phase, instead, the information about the change in the topology is propagated to all the other nodes in the network (this second phase also resolves possible routing loops that can be introduced by the detection phase). Once the propagation phase in completed, all the nodes have the same information base and can take consistent routing decisions.

More in details, the failure detection phase happens thanks to *HELLO* messages, which are messages generated periodically by every node of the network and transmitted in broadcast to the 1-hop

neighbors. The periodicity of *HELLO* messages depends on a timer whose value is the same for every node of the network when standard OLSR is used (RFC 3626 suggests to use a 2 seconds timer). Each *HELLO* message contains a validity field whose value is a configurable multiple (*HELLO* validity multiplier) of the generation timer. Every time a node receives a *HELLO* message from a neighbor, it set a validity timer to the value contained in the validity field. If a new *HELLO* message is not received from the same neighbor before the validity timer expiration, the link towards that particular neighbor is considered broken (link failure detection). Also, *HELLO* messages are used in OLSR for estimating the link loss and for assigning a link quality using the ETX metric. Obviously, the link qualities estimated using *HELLO* messages affect the choice of the shortest paths.

In a similar way, the propagation phase happens thanks to Topology Control (*TC*) messages, which are messages generated periodically by every node of the network and flooded to reach every other node so that every node is aware of the full (weighted) topology and can compute the shortest path to any destination and build its routing tables. As in the case of *HELLO* messages, also the periodicity of *TC* messages depends on a timer whose value is the same for every node of the network when standard OLSR is used (RFC 3626 suggests to use a 5 seconds timer) and each TC message has an associated validity timer whose value is a configurable multiple (*TC* validity multiplier) of the generation timer.

From the description above it should be clear that the lower the timers used to generate control messages (especially *HELLO* messages), the faster is the protocol in detecting a link failure and in distributing the correct network information to all the nodes. However, the timers used by a node can not be arbitrarily low because control messages increases the overhead and subtract resources to useful network traffic. Pop-Routing is designed to solve this problem and to find a trade-off between protocol convergence speed and overhead generated by control messages: it performs an optimal equalization of the control messages generation timers that maintains the total overhead constant and minimizes the damage produced by a node failure. In particular, with Pop-Routing the *HELLO* and *TC* generation timers are not the same for every node. Instead, they are dynamically computed by a function that depends on the betweenness centrality of a node, where the betweenness centrality is a graph metric that represents the fraction of all the shortest paths that pass through a single node and can be seen as an indicator of the importance of that node in the network. The rationale of Pop-Routing is that the failure of a node with high betweenness centrality triggers the re-computation of a large number of shortest paths, and thus possibly impacts a high amount of traffic: it is therefore convenient to decrease control messages generation timers in order to quickly detect the failure of that node and quickly propagate this information to the rest of the network. Instead, the failure of a node with low centrality has a low impact and so large values for *HELLO* and *TC* generation timers can be use. We implemented the Pop-Routing mechanism in an open source software called Prince whose architecture and interaction with OLSRd is explained in more details in section B.3.

B.1.2 Background and Motivation

The goal of these experiments is twofold. First, we want to test the validity of Pop-Routing in a real testbed. So far, we only tested Pop-Routing through simulations and emulations, where we were able to set up a perfectly controlled environment. We were still lacking a “real world” test and the WiSHFUL OpenCall 3 was perfectly fitting our purposes. Second, we want to understand the impact of different settings and parameters, which only have an impact in reality. As an example, consider the Prince update interval (Prince is better described later on). Prince periodically fetches the topology from OLSR and computes the timers. Depending on how frequently this is performed, we might have different outcomes. In a controlled environment, this is a non-problem, in a real

environment, instead, we found that this timer has important implications in the interactions with the OLSR protocol, as we explain for Fisheye in the next section.

B.1.3 Set-up of the Experiment

The designed experiments aim at reproducing a real-world wireless ad-hoc/mesh network for testing our optimized version of the OLSR routing protocol. Setting up the experiments requires different steps, which include:

1. Setting up the nodes;
2. Configuring the nodes; and
3. Setting up and running the experiment.

Point 1 includes copying our control scripts on all nodes, as well as downloading and installing the required software (e.g., the OLSR daemon and Prince, our plugin that implements Pop-Routing). In addition, the scripts download and install the WiSHFUL framework. This is automatically performed using *ansible*, a tool that permits to execute multiple tasks in parallel on a cluster of nodes through SSH.

Point 2 deals with network setup. We automatically choose a WiFi card and create an interface that is configured depending on our needs. In this particular case, we setup an ad-hoc network on a specific channel, using a desired PHY-layer bitrate and transmission power. We mainly resorted to a robust modulation and coding scheme (BPSK R=1/2, 6 Mbps) and the highest possible power to create a full-mesh, which is required for the third point. Finally, the setup automatically assigns an IP address to the ad-hoc interface depending on the hostname (e.g., the node named *nuc0-12* is assigned with the IP *10.1.0.12*). At this point, the nodes are already capable of communicating. To set up the ad-hoc network we use three UPIs: start_adhoc(), set_modulation_rate(), and set_tx_power(). The start_adhoc() UPI is not officially in the WiSHFUL documentation, but its interface is defined in the source code. To be able to use it, we had to modify its interface, as this was defined without parameters.

Point 3 runs the experiment. The first step is creating the topology. We do this by means of *iptables* rules, filtering packets at the MAC layer (Figure 1). Our scripts can reproduce a topology described in a *.graphml* file or generate synthetic ones with certain characteristics, depending on configuration parameters. The script first starts the OLSR daemon with filtering rules disabled. This permits us to fetch the testbed topology by querying OLSR. Using the obtained topology, we “implement” the desired one by selectively disabling links. Clearly, the closer is the base topology to a full-mesh, the easier is to map the experiment topology on the real one, which is why we are using a robust modulation and coding scheme and a high transmission power. The second step is starting OLSR on the desired topology and wait for convergence. To do so, we check that the topology and the routing table in a specific node do not change for a certain period of time. Once reaching convergence, we start our logging module on all nodes in a synchronized fashion using the Network Time Protocol. The aim of the module is to dump for each node the routing table and the topology as seen by OLSR during time, typically taking a sample every few hundreds milliseconds. Finally, we schedule a change in the topology, which is implemented by killing one or more nodes at a certain point in time to see how OLSR reacts to the event. The node-killing policy can be chosen through a parameter of the experiment. For example, we can tell the scripts to kill the most central node (in terms of betweenness centrality), the second most central, etc. Obtaining the test-bed topology and “implementing” the experiment topology is performed by using a set of UPIs. First, we setup the ad-hoc network using the start_adhoc(), set_tx_power(), and set_modulation_rate() methods. We then use the start_local_control_program() UPI to perform some operations that are very specific to

our experiment. For example, we can start the OLSR daemon, we can create a specific configuration file for it, and we can query the current topology and routing table. For the future, we might think about creating a new set of generic routing UPIs (e.g., start_routing_daemon(), stop_routing_daemon()) which could then be implemented in different WiSHFUL modules, i.e., one per routing protocol (e.g., WishfulOLSRv1Module, WishfulOLSRv2Module, WishfulRIPModule, etc.). Finally, after computing the set of iptables rules to be applied, we use the get_iface_hw_addr() and the filter_mac() UPIs. The get_iface_hw_addr() is a standard WiSHFUL UPIs that retrieves the MAC address of an interface of a node. The filter_mac() UPI, instead, is a new UPI we propose and we implement that performs MAC layer filtering through iptables. The UPI takes the MAC address to filter as argument. The reason for implementing a new one is that existing UPIs for filtering work at the IP layer. This does not fit our purposes, as we want to disable point-to-point links, but not the communication at the IP layer.

To run the experiment, each participating node runs as WiSHFUL agent, while one of the nodes acts as a WiSFHUL controller (our master node). The controller loads the experiment parameters from a YAML file, a feature that comes for free by following the WiSHFUL agent-controller philosophy. An example configuration file is shown below. We are not going to describe all the parameters. The important aspect here is that we can explore a wide range of scenarios by simply changing values in this configuration file.

```
controller:
    name: "Controller"
    info: "Poprow Controller"
    dl: "tcp://10.0.3.43:8990"
    ul: "tcp://10.0.3.43:8989"
    power: 20
    channel: 2412
    rate: 6
    nodes: 4
    metrics_seed: 0
    single_interface: false
    fixed_intervals: false
    hello_mult: 10
    tc_mult: 60
    disable_lq: false
    graph_params: "random_regular_graph:d=2,seed=0"
    metrics_in_graph: false
    kill_strategy: "stop_mostcentral_1s_repeat"
    strategy_params: "0"
    repetitions: 10
    exp_name: "t500"
    testbed: "wilab"
    resultsdir: "/proj/wall2-ilabt-iminds-be/exp/wilabolsr/"

modules:
    discovery:
        module : wishful_module_discovery_pyre
```

```
        class_name : PyreDiscoveryControllerModule
                                kwargs:    {"iface":"eth0",    "groupName":"poprow",
"downlink":"tcp://10.0.3.43:8990", "uplink":"tcp://10.0.3.43:8989"}
```

Each experiment is run with standard OLSR first, and then with OLSR and Prince. In addition, each experiment is repeated multiple times to obtain statistical confidence. When all experiments are completed, the script downloads the results of the experiment on our local machines for data processing. During the data post-processing phase we compute the number of non-functioning paths (broken and looped paths) by collecting the timestamped routing tables from all the nodes of the topology and by navigating them for each source and each destination in all the dump points. For a given sampling point, a path is considered broken when the routing table of one of the node along the route points to the killed node or does not have an entry for the destination. In a real running network these paths would generate an ICMP "*Destination Unreachable*" message. Looped paths, instead, are the ones where there is a node that appears twice, causing packets to bounce back and forth until the Time to Live (TTL) expires. In a running network they generate ICMP "*Time Exceeded*" messages. As shown in more details in Section B.2, Pop-Routing performance are compared against standard OLSR by analyzing how many routes are not-functioning and for how much time. More formally, let *e* be the total number of samples (timestamped routing tables for each node), r_h and T_h the number of non-functioning paths and the time-stamp of sample *h*, respectively. We define the "combined empirical loss reduction" *L* as

$$L = \sum_{h=1}^{e} r_h \cdot (T_h - T_{h-1})$$

which can be seen as the integral of the number of non- functioning paths over time.

To compare OLSR with Pop-Routing, we compute the absolute integral difference $L_{olsr} - L_{pop}$ for each of the failed nodes. In addition, we compute the global absolute and relative loss reduction as

$$L_A = \sum_{i=1}^{N_f} (L_{olsr}(i) - L_{pop}(i)),\ L_g = 1 - \sum_{i=1}^{N_f} L_{pop}(i) \,/\, \sum_{i=1}^{N_f} L_{olsr}(i)$$

where $L(i)$ indicates the loss reduction computed when killing the i-th most central node that is neither a cut-point nor a leaf and N_f is the number of experiments.

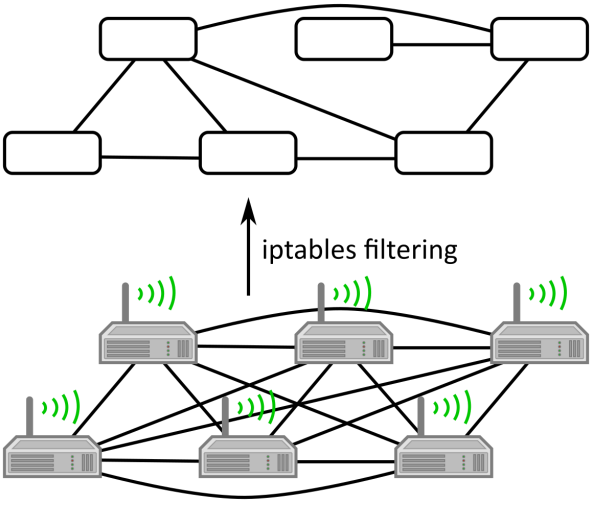


Figure 1: Creation of a topology through *iptables* filtering.

In the experiments we use several different topologies to work with, some of them because they are simple and help understanding and explaining results, others because are of widespread use as models, and some others yet because are highly realistic. In general, all topologies exploit all the available nodes in the testbed (42) to have the largest possible network, only in some cases this is not possible. We choose the topology in Figure 2 to test the expected behavior and to run functional tests. We refer to this with the name of *shortcut* topology. It is a chain which splits in two paths, one with an even and one with an odd number of nodes. The length of the left and right chains is configurable. The idea is that all the shortest paths going from the left/right chain to the right/left chain pass through the shortest of the two alternative paths. When killing a node inside the shortest alternative paths, the routes should be reconfigured to pass through the longest path. Being the nodes in the shortest alternative path very central due to the number of shortest paths passing through them, we expect Pop-Routing to be faster in convergence than the vanilla version of OLSR.

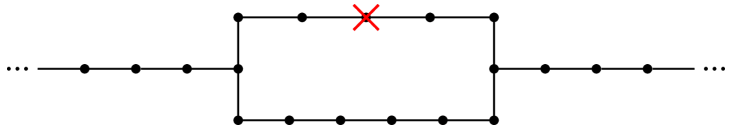


Figure 2: Shortcut topology (odd path of length 5, even path of length 6).

A second synthetic topology is generated using a Barabasi-Albert model. The model follows a preferential attachment mechanism, i.e., during the generation phase a new node is linked to *m* existing nodes chosen at random, and the probability of choosing a certain node depends on its degree. The higher the degree, the higher its probability of being chosen. In our experiments, we consider *m = 3*.

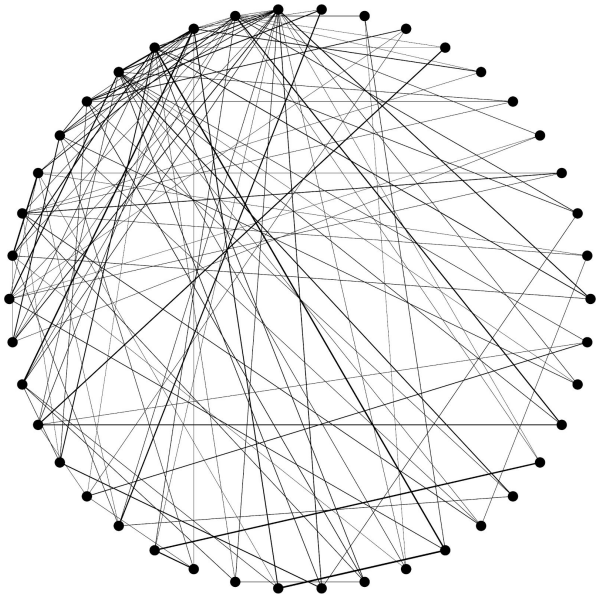


Figure 3: Barabasi-Albert random graph with *m = 3*.

We then generated a topology that resembles a backbone (Figure 4). The idea is to have 6 fully-meshed core node and a set of peripheral nodes, each of them connected to two core nodes.

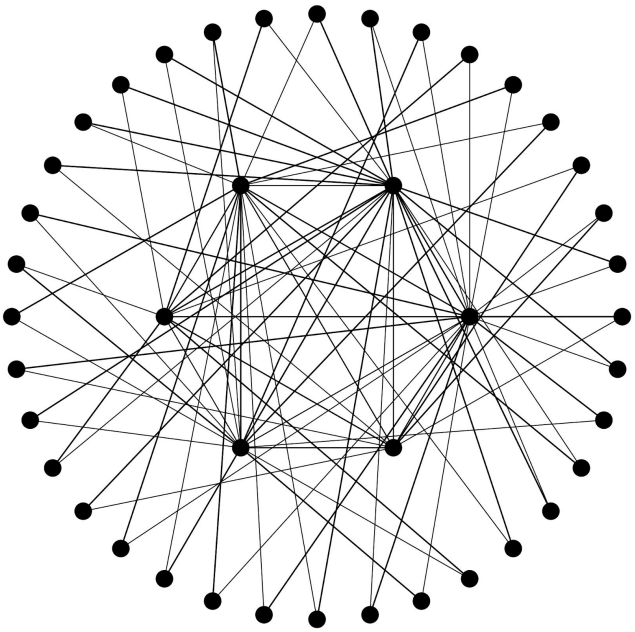


Figure 4: Backbone topology with 6 core nodes.

Moving to realistic topologies, we considered two well-known topology types with 40 nodes: caveman (Figure 5) and Waxman (Figure 6). The former is characterized by multiple, well-connected cliques, inter-connected by rewiring a few links and is generally used to model human communities, and thus, ad-hoc networks. The latter is often used to model communication networks.

Killing nodes

The goal of the experiments is to measure the time needed to recover all broken and looped paths after a failure, so we have to kill selected nodes and collect routing tables in the aftermath. Depending on the topology, we use different strategies for selecting the nodes to kill. For the line graph, we kill the node in the middle of the shortest branch. In the Barabási-Albert topology, we kill the node with the highest betweenness centrality. In the caveman and the Waxman topology, we kill the five nodes with the highest and the lowest betweenness centrality.

To kill a node, we propose and implement a new UPI, i.e., interface_down(), which disables the wireless interface used by OLSR, simulating the failure of the node. We use this UPI with the exec_time() directive, permitting us to kill the node at a specific time during the experiment.

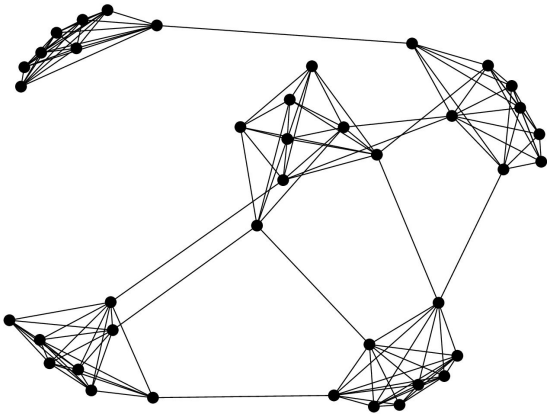


Figure 4: Caveman topology.

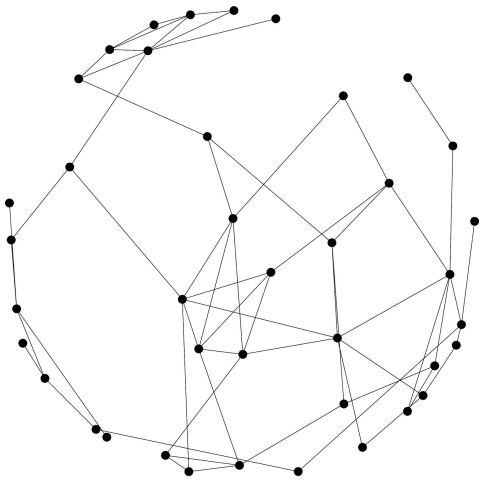


Figure 5: Waxman topology.

B.2 Technical Results & Lessons learned

The first set of experiments was aimed at understanding the correct parameters for the OLSR daemon and the experiment configuration. Initially, the setup had severe problems with respect to convergence, meaning that either the network was never converging or that in different runs of the same experiment the same logic topology converged to different shortest paths between some pairs of source/destination nodes. The first case makes it impossible to actually run the experiments, while the second makes runs incomparable.

Before introducing actual results, we describe a typical experiment output for the sake of clarity. Figure 7 shows the typical evolution of an experiment after killing one node, for standard OLSR and for Pop-Routing. For both OLSR and Pop-Routing, killing a node breaks all the shortest paths passing through that node. As the topology is the same, the number of broken routes is equal. After a few seconds the neighbors of the killed node update the routing table by removing the killed node as a possible next hop. This causes the number of broken routes to decrease quickly but, on the other hand, creates some routing loops. These loops are removed when TC messages are sent and the information about the topology change is propagated.

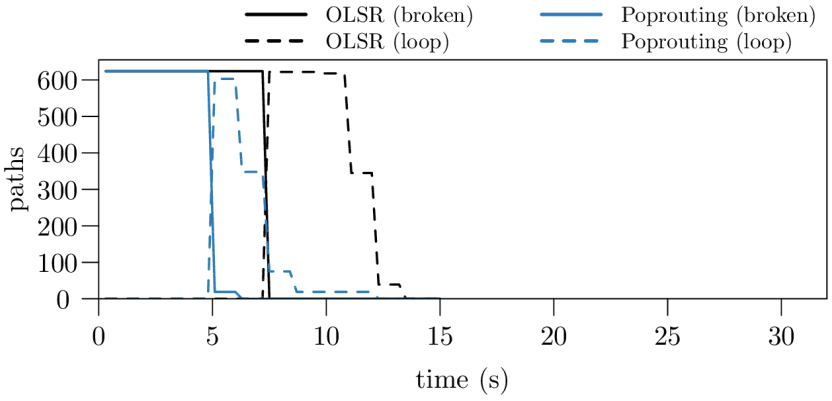


Figure 6: Typical experiment run. Shortcut topology, Hello and TC validity multipliers set to 3 and 60, respectively.

We now describe some of the issues we faced during the first test experiments. Our analysis highlighted that convergence problems are caused by a combination of factors. One OLSR feature

causing problems is the Fisheye routing mechanism. Fisheye is a mechanism that reduces the overhead of the TC flooding mechanism in portions of the network that are far from the origination point. This is realized by changing the TTL value used in successive TC packets. As an example, the standard implementation uses the sequence 2, 8, 2, 16, 2, 8, 2 and 255. So the first TC message generated from a certain node is received by all its 2-hop neighbors, the second by its 8-hop neighbors, and so on. This means that the whole network is flooded with the message from a TC originator only every 8 TC intervals. This can cause a problem when the diameter of the network is large and when the TC validity is small. Indeed, far nodes will miss each other TC messages for long periods of times, causing the wrong invalidation of links. Then, when a node uses a TTL value of 255, the invalidated links will be considered again by all nodes in the network. This is made even worse by Pop-Routing, which periodically changes the timers and makes the system even more unstable. Figure 8 shows the evolution of an experiment with Fisheye routing enabled for the shortcut topology, which is a large-diameter network (more than 30 hops). In the first 20 seconds we have roughly 500 routes broken due to the node failure. Then both the standard and the optimized versions of OLSR fix the routes. However, Pop-Routing never converges, while standard OLSR invalidates some routes around 260/270 seconds. Clearly, this makes it impossible to obtain meaningful results, so we had to disable this feature. Actually, Pop-Routing superseeds Fisheye, so their combined use would have not been advantageous anyway.

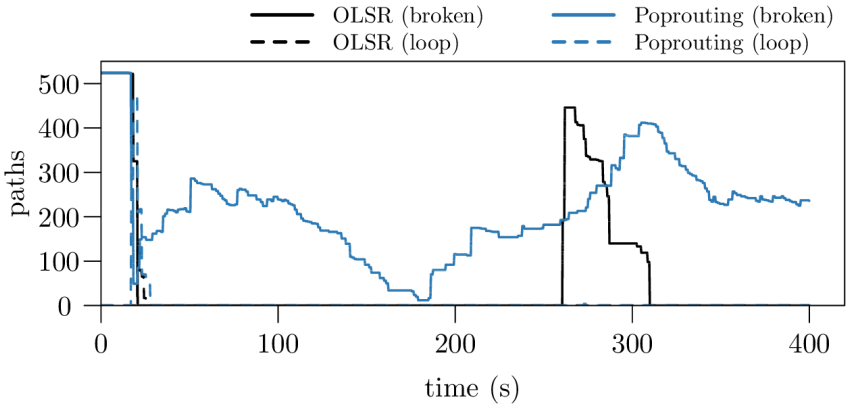


Figure 7: Network instability caused by Fisheye routing. Shortcut topology, Hello and TC validity multipliers both set to 10.

The next parameters we investigated are the Hello and TC validity multipliers. We discovered that a proper value is a trade off between reactivity to changes and stability. As an example, compare Figures 6 and 8, which both show route disruption evolution for the shortcut topology, but with different multipliers. In Figure 6, the Hello multiplier was set to 3, while in Figure 8 to 10. Qualitatively, the behavior is similar, but in Figure 6 it takes much less time to resolve routing issues. However, highly aggressive multiplier values might generate false positives, which wrongly invalidates network links. Figure 9 shows an example. After having partially re-converged following node failure, OLSR continuously invalidates routes and tries to re-converge again, but apparently it never succeeds.

This behavior is due to the fact that our logical topology is built on top of a fully-meshed physical network. Every frame generates interference to all nodes in the testbed, increasing the chance of frame collision. Given that OLSR Hello and TC packets are broadcast there is no acknowledgement, and colliding frames are simply lost. If we consider long flooding paths between nodes at the edges

of the network, the chances of multiple consecutive losses are high. For this reason, in our experiments we decided to use the multipliers used in the OLSRd daemon implementation (10 for Hello, 60 for TC) instead of the ones proposed in the standard (3 for Hello, 3 for TC). This observation, that has never been reported in the literature to the best of our knowledge, is very interesting and deserves further investigation and research. Indeed, the default multipliers set by the OLSRd daemon are way too large to ensure good performance; however, we are sure they were set so large based on observations similar to ours. This means that probably the well consolidated way of proof-of-existence, based on receiving HELLOs regularly is not well suited for some environments, possibly even for all wireless meshes, at least those using 802.11 as basic technology. We will further investigate on this theme after the end of the project.

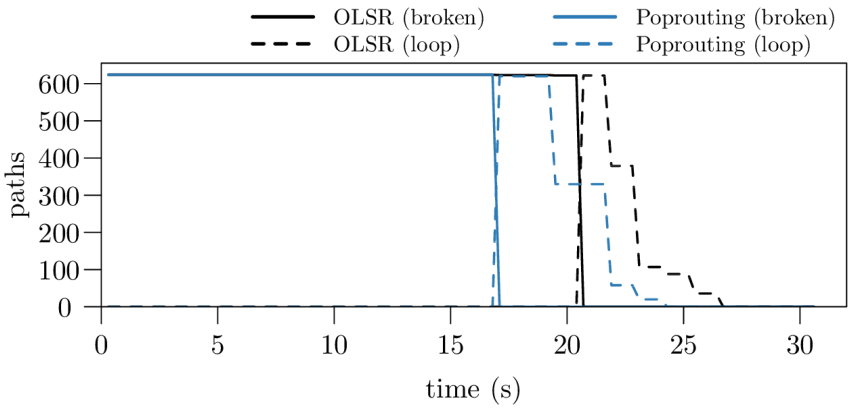


Figure 8: Shortcut topology, Hello and TC validity multipliers set to 10 and 60, respectively.

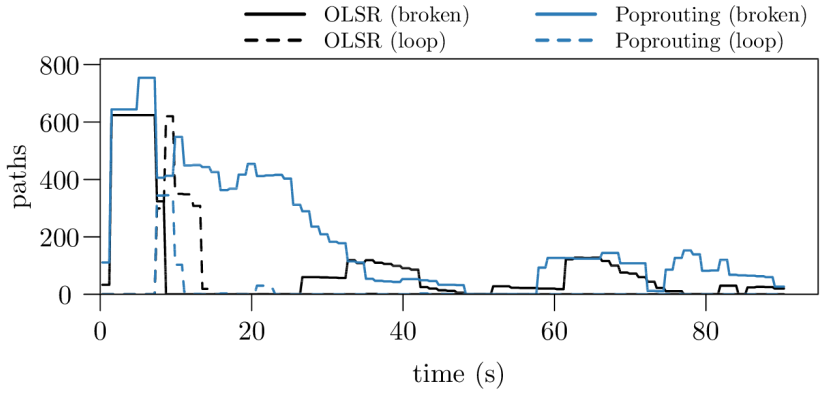


Figure 9: Shortcut topology, Hello and TC validity multipliers both set to 3.

An additional problem we encountered was due to the existence of multiple shortest paths between certain pairs of nodes. When having multiple, same-length shortest paths, the choice of the best one is down to the link quality estimation of OLSR. In the fully-meshed w.iLab1 test bed, link qualities are very similar, so slight variations might cause OLSR to converge to a totally different routing topology. This can causes experiments to be incomparable. In one experiment we might have a certain number of paths passing through a node, while in another the situation might be completely different. One example topology for which we faced this problem is a shortcut topology similar to the one in Figure

2, but where the two branches have the same length. In that case the choice of the shortest paths going from the left chain to the right and vice versa changes completely from one run to the other. When killing a node in one of the two paths, in one run more than half of the paths can get broken, while in another only a few depending on which branch OLSR prefers for routing.

We solved this problem by using OLSR link quality multipliers: for each link in the topology we generate a random multiplier in the range [0.5, 1] that penalizes the link. This way we reduce the probability of having multiple same-cost shortest paths. This does not completely solve the problem, but it reduces its severity. Figure 10 shows two different runs of the caveman topology. In one case, we have more routes passing through the killed node in the vanilla OLSR experiment, while in the other we have more routes in the Pop-Routing one. The difference is however minor, so averaging over multiple experiment eliminates the problem.

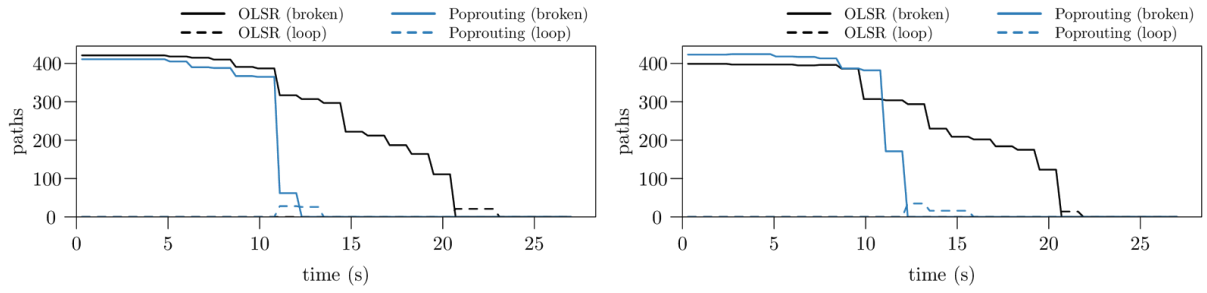


Figure 10: Network convergence discrepancies due to multiple possible equivalent shortest paths where OLRS chooses at random based on small fluctuations of the link quality, the issue is shown clearly by the different number of shortest paths that are immediately affected by the failure.

The final issue we observed in the experiment is the unexpected increase of broken paths during the recovery phase. We expect in fact the number of broken paths to be always decreasing. Figure 11 shows this problem around times 15 and 18 seconds. We discovered that this is also due to little link quality oscillations. Consider Figure 12 for an explanation. Imagine the red node to be the failed one, and that the current best routing path between the north and the south clouds is the green one. If any of the link qualities on the green path degrades and the nodes are not yet aware that the red one is down, they might decide to switch to the left path. The path is however broken, so some paths that were considered as valid are now invalid, increasing the number of broken paths in our plots. Unfortunately, there is no solution to this problem, so the only way to get rid of this is by averaging multiple repetitions.

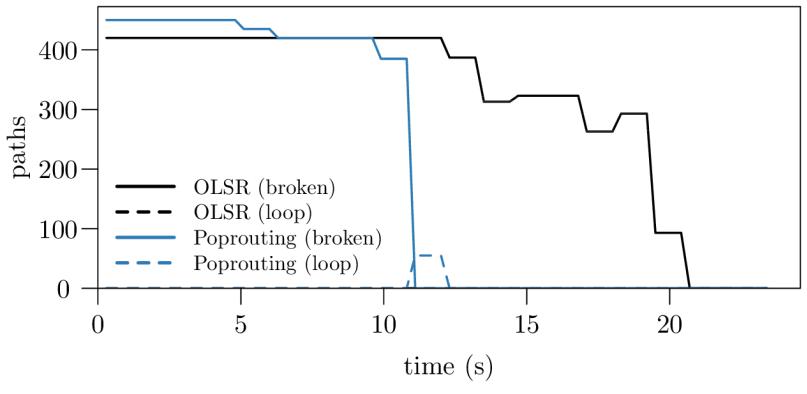


Figure 11: Increasing number of broken paths due to route switching.

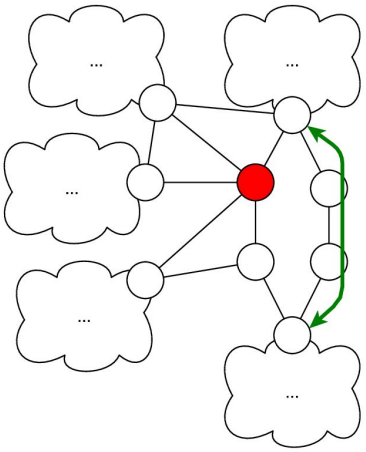


Figure 12: Wrong route switching problem.

Figure 13 reports an interesting behavior observable when multiple paths from a source to a destination exist, like in a Barabási-Albert model. The figure shows that the network starts to re-converge before the expiration of the H validity time. This happens because OLSRd implements the ETX metric, which causes link quality to decrease for each missed H message. Even before the link is considered broken its quality degrades, and this information is propagated with TCs. Nodes progressively decide to switch to other shortest paths, and the dead node becomes less central. Pop-Routing still performs better than OLSRd, but the gain is smaller. This phenomenon is not observable in Figures 6 and 8, as in this specific scenario the cost of the additional hop makes the shorter path preferable until the node failure is discovered.

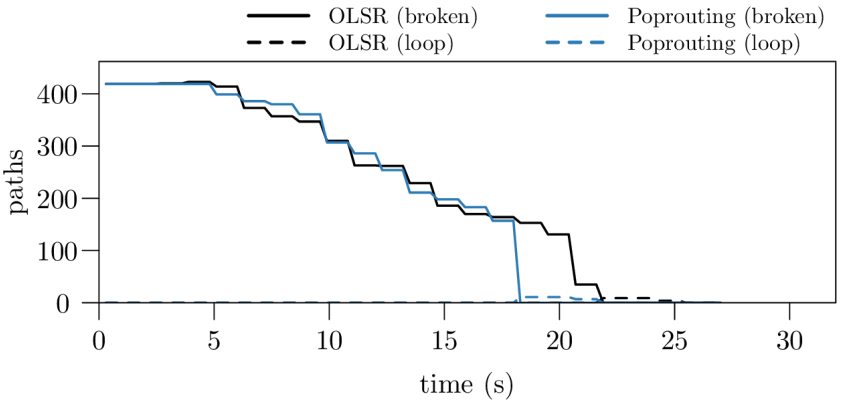


Figure 13: Stairs-like behavior due to link quality degradation in a Barabási-Albert graph (TC multiplier set to 10).

We continue the analysis considering the caveman and the Waxman topology types. Figure 14 shows the computed timers and the betweenness centrality for a single run using the Waxman topology. The nodes are ordered by decreasing values of betweenness. Points drawn in light red are cut-points and leaves, i.e., the ones that can not be killed without partitioning the network. In addition, the plot

shows the default timer values for comparison. The plot shows that most central nodes are assigned lower timers with respect to the standard values. On the other hand, least central nodes are penalized and are assigned higher timer values.

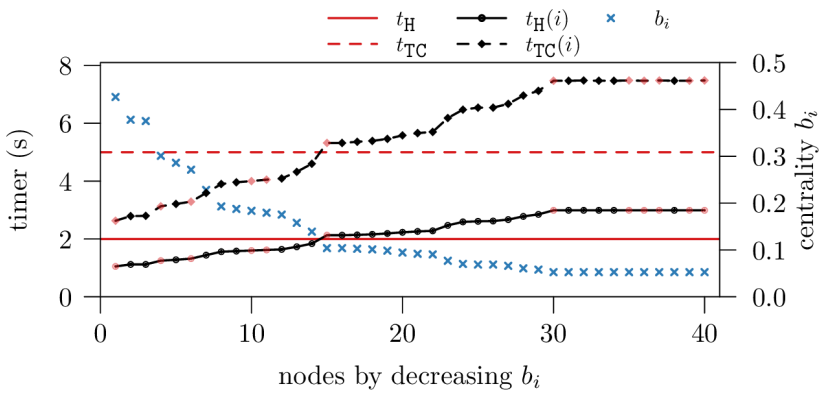


Figure 14: Computed timers and betweenness centrality for one run using the Waxman topology. Light red dots identify nodes that failing would partition the network.

Figure 15 shows the absolute integral difference for all killed nodes. The node index does not consider cut-points and leaves. Pop-Routing shows a positive performance gain with respect to OLSRd when killing the five most central nodes. When killing the five least central nodes (negative node index), in most cases the performance loss is null, as there is no shortest path passing through the failing node. In the remaining cases, the loss is negligible compared to the gain obtained by increasing the Hello and TC message frequency for the most central nodes.

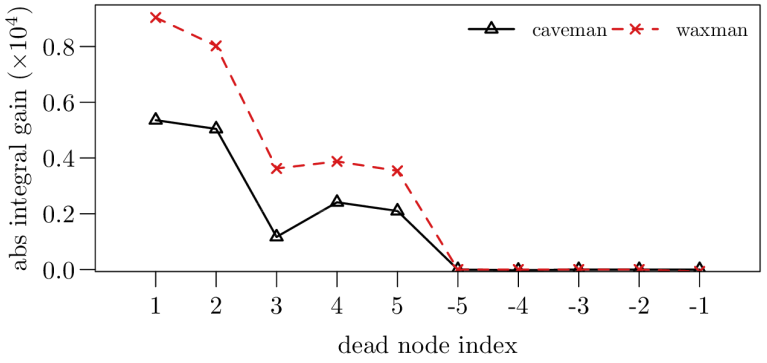


Figure 15: Absolute integral gain as function of the killed node index, for two different topology and two different interpretations of the degree. Nodes are ordered by betweenness centrality. Negative indexes indicates the least central nodes.

We would expect the absolute performance gain to be monotonically decreasing with the node index, as the timers in Figure 14 are monotonically decreasing. The values in Figure 15, however, are heavily affected by noise as the shortest paths might change from one experiment run to the next. Indeed, as witnessed by Figure 14, the two most central nodes have a very similar betweenness centrality value, so slight fluctuations in signal quality can lead to route changing and affect the results.

To obtain a complete comparison we would need to repeat the experiments killing each possible node; however, this required a total time that we did not have before the end of the project (we planned to continue running experiments in the future). As an alternative, we complement experiments with the theoretical gain that Pop-Routing can achieve on a certain network (the original paper offers a formula to compute the theoretical gain analytically). Table 1 reports the theoretic gain computed considering all nodes. The gain is smaller than when considering the five most and least central nodes as well, but it is in any case positive, showing a performance improvement by 10% and 4% for the Waxman and the caveman topology types, respectively, with no additional protocol overhead. It is interesting that the experimental gain is larger than the theoretical when we consider the same modes. This is probably due to noise in experiments, as already mentioned, but its interpretation requires additional investigation.

To improve the overall performance of Pop-Routing we analyze cut-points. Cut-points are nodes that cannot be removed without partitioning the network. A bi-connected component instead is a sub-graph of the network for which the removal of one node will not partition the network. Any graph can be decomposed in a so-called block-cut tree made of biconnected components and cut-points, as shown in Figure 16. This representation shows that typically cut-points have high betweenness, and thus we expect their timers to be lower than the timers of other nodes.

On the other hand, if a cut-point fails the network is physically partitioned, and the routing protocol can not fix it. That is the reason why in our experiments we never kill a cut-point, because some routes would never converge, and the network before and after the failure would not be comparable.

In general for node n_i, knowing that a piece of the network is not reachable anymore is not particularly useful: if n_i is exchanging traffic with n_j and due to a failure there exist no path anymore from n_i to n_j applications stop working, and routing cannot do anything about it. If we look at Pop-Routing as an optimization strategy, given a certain amount of a resource (the airtime devoted to control messages) Pop-Routing distributes it among nodes (tuning their timers) to reduce the outage due to failures. The problem is that Pop-Routing assigns resources to cut-points, but their failure can not be fixed.

A straightforward conclusion would be to artificially treat cut-points as nodes with minimum centrality and distribute their resources to other nodes. This is not entirely correct, because part of the centrality of a cut-point is due to the shortest paths that connect nodes in the same block. If we do not consider this we ignore the fact that, when the cut-point fails, some of the shortest paths that pass though it can actually be fixed. The correct decision would be to compute centrality for cut-points considering only the shortest paths whose endpoints are in the same block, and ignore the paths that have both endpoints in different blocks. More formally, if n_k is a cut-point belonging to a number l of bi-connected components $\{B_1, \dots, B_l\}$, N_h the size of each bi-connected component, and b_k^h the betweenness centrality of n_k in B_h computed using the standard method, then its modified betweenness $b_k^{'}$ is defined as:

$$b_k^{'} = \frac{1}{N(N-1)} \left(\sum_{h=1}^{l} b_k^h (N_h(N_h - 1)) + 2 \cdot (N - \sum_{h=1}^{l} N_h) \right)$$

where the multiplication by $(N_h(N_h - 1))$ is needed to remove the normalization applied by the standard betweenness centrality and the term $2 \cdot (N - \sum_h N_h)$ takes into account the shortest paths that have as endpoints n_k and any other node $n_i \notin B_h$ when n_k is considered in the component B_h. This is a further optimization we studied from the observations done in POPROW that we will introduce in Prince in the future and that we plan to analyze using WiSHFUL test beds. We can

anyhow compute the timers and the theoretical expected gain, which are reported in Figures 17 and 18, and Table 2, respectively. By comparing these values with the ones in Table 1 a clear further advantage emerges.

One exception to this rule that requires a special treatment would be the case of gateways. When the block containing a gateway is not reachable anymore, then it is crucial to know it as soon as possible, in order to switch to another working gateway (if it exists). This special case will be treated directly in the implementation in Prince.

	Waxman	Caveman
Absolute	2803.31	1606.06
Relative	0.26	0.21
Relative (theory)	0.18	0.20
Relative (theory, all nodes)	0.10	0.04

Table 1: Absolute and relative gains for Waxman and caveman topology.

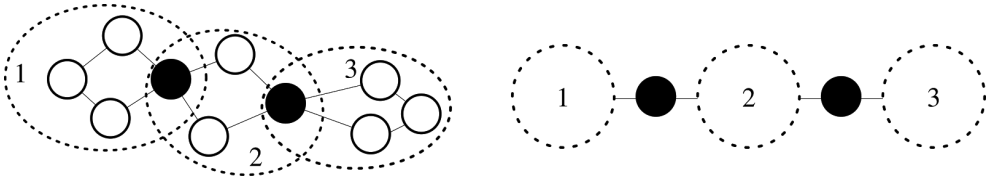


Figure 16: An example decomposition in a block-cut tree.

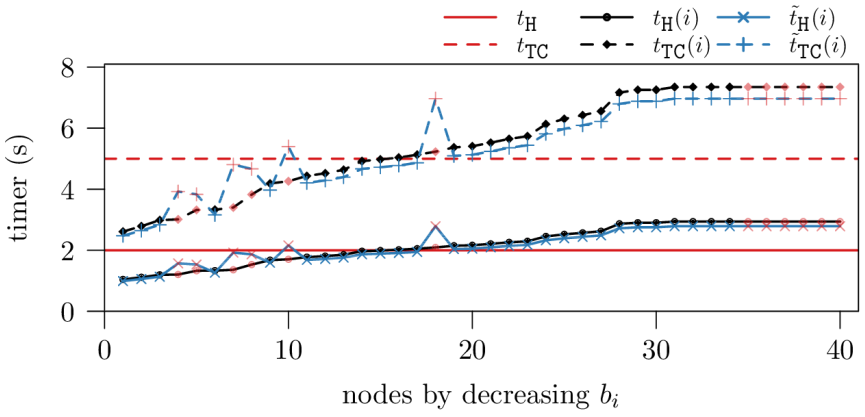


Figure 17: Timers comparison between the base Pop-Routing algorithm and the ones considering a different centrality for cut-points for the Waxman topology.

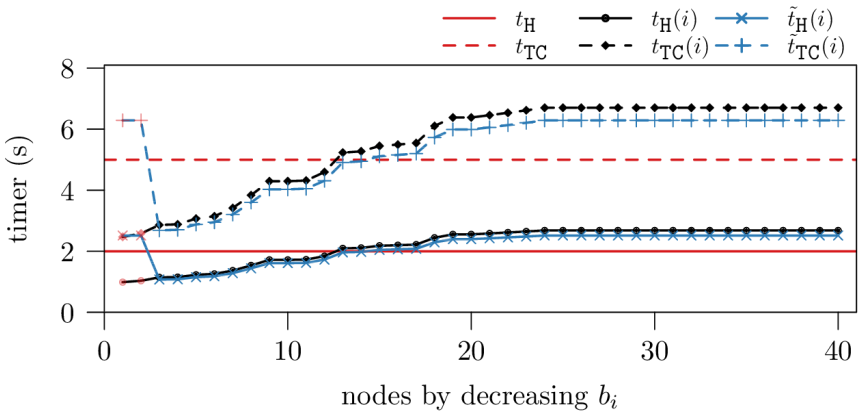


Figure 18: Timers comparison between the base Pop-Routing algorithm and the ones considering a different centrality for cut-points for the caveman topology.

	Waxman	Caveman
Relative (theory, all nodes)	0.14	0.10

Table 2: Absolute and relative gains for Waxman and caveman topology with modified centrality.

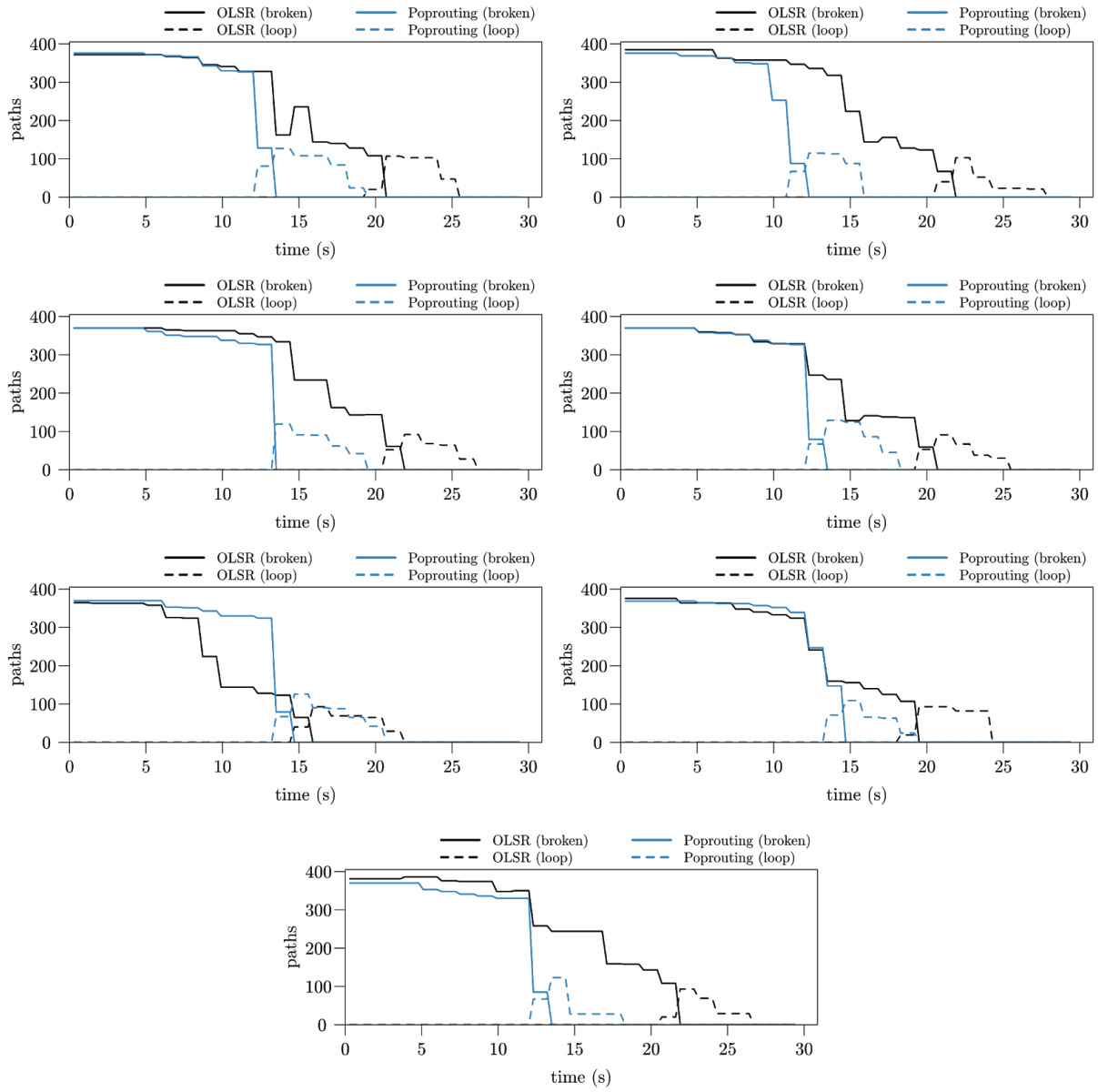


Figure 19: Seven repetitions on the Waxman topology when killing the second most central node.

Finally, Figure 19 reports the comparison of OLSR and Prince on a batch of 7 runs when killing the same node (node 2 in Figure 15). We report this set of graphs in order to outline the difficulties that emerge when testing Pop-Routing, at every run the results are different: the network converges in a different time, the convergence curve is different, the impact of the loops is different. The value of the WiSHFUL testbed was to show us these differences and find reasonable countermeasures for the configuration of Prince.

B.3 Prince software architecture

Prince, whose architecture and interaction with OLSRd are illustrated in Figure 20, is the open source software implementation of the Pop-Routing mechanism.

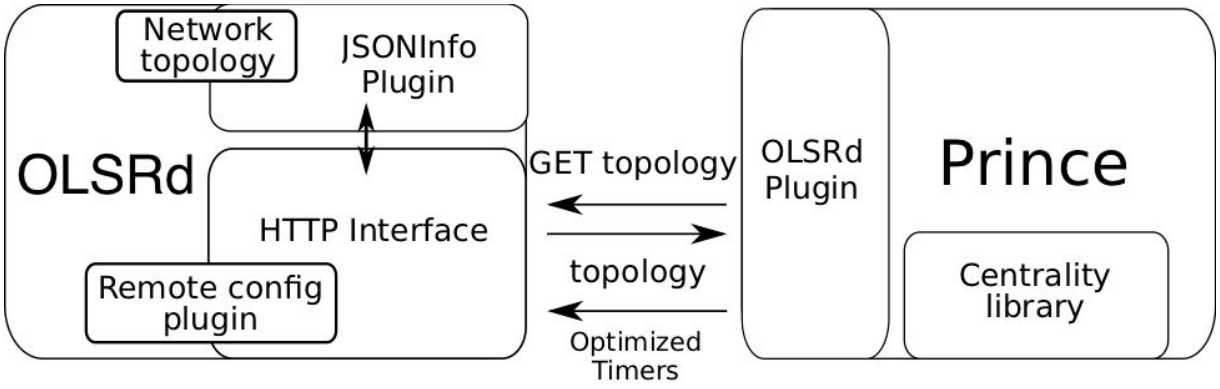


Figure 20: Interaction between Prince daemon and OLSRd

Prince is executed on each node of the network and exploits the "HTTP interface" made available by OLSRd (the daemon that implements the OLSR protocol) to access the other plugins of OLSRd and periodically receive the network topology, calculate the correct value of the generation timers and feed them back to the running instance of OLSRd. In more details, every configurable number of seconds (usually in the order of minutes), Prince send an HTTP request to OLSRd "HTTP interface" to access the JSONInfo plugin in order to ask for an updated version of the network topology. Prince receives back the current network topology in a JSON format and gives it as input to the Centrality library whose responsibility is to compute the betweenness centrality of every node of the network. Once the Centrality library completed its execution, Prince can use the betweenness centrality value of the local node for computing the optimal values of the generation timers for the *HELLO* and *TC* messages. Finally, using again the "HTTP interface", Prince sends the optimized timers values to the Remote config plugin that allows changing the OLSRd configuration parameters at runtime. It is important to note that Prince is a standalone daemon separated from OLSRd and it is designed to run asynchronously with respect to OLSRd. This design choice was necessary because the computation of the betweenness centrality values done by the Centrality library may take several seconds in large networks on constrained device, and OLSRd can not freeze for such an amount of time. Another feature of Prince is that it does not interface directly with the routing protocol, instead it exploits a plug-in system (OLSRd plugin in case of OLSRd) that makes it very easy to extend Prince to support any other routing protocol capable of exporting network topology information. In fact, the current version of Prince already supports also OLSRv2 through the plug-in OONF. Also, given that OLSRv2 shares the same basic functions with OLSR, the Pop-Routing principle could be applied to OLSRv2 as well without any modifications. Unfortunately, the daemon implementation of OLSRv2 is still under heavy development and its current development state is not mature enough to be reliably tested in real networks.

The POPROW project turned out to be useful not only for evaluating the Pop-Routing principle in real networks, but also for improving the software that implements Pop-Routing (both Prince and Remote config plugin). In fact we upgraded the logging capability of Prince making it easier to debug problems during experiments and improving our capability of performing post-processing data analysis. We added the support for both directed and undirected graph in the JSON parser of Prince solving a few incompatibilities with some implementation of the JSONInfo plugin. Finally, we also extended the Remote config plugin which now allows to configure also the HELLO and TC validity multipliers, functionality that can be useful to adapt OLSRd to different network types.

Prince software is Open Source and public at the github node of the ANS research group:

https://github.com/AdvancedNetworkingSystems

B.4 Impact

The research on large mesh networks has been thriving in the last years, at least 4 FP7/H2020 European research projects fully or partly deal with CNs (see the netCommons project, the CONFINE, the CLOMMUNITY and the P2P-Value project), and there is a general technical and socio-technical interest in CNs, that are a promising technology to address underserved and digitally-divided areas[1] with a bottom-up approach.

The results of POPROW were key to understand how far Prince is from its deployment in a real network. We have observed that while OLRSd is a stable routing daemon, its new version OONF (implementing the second version of OLSR) is not ready yet for production, and thus we focused on the stable OLSRd. While testing in POPROW, several issues that were so-far not clear to us emerged and opened new challenges for this research. Here we just mention three of them.

1) In the networks we observed a small impact of the temporary routing loops, which can be due to the small size of the network, or to the fact that realistic topologies are not subject to such events. We need further investigation to understand how Pop-Routing should invest into optimizing TC messages to fix routing loops.
2) The impact of quality metrics in the route convergence is higher than we expected in certain network scenarios. In some cases shortest paths get smoothly corrected due to the propagation of TC messages even before the link is declared dead. This again is something we did not evaluate correctly in emulation before POPROW and gives us fresh input to improve Pop-Routing
3) The special treatment of cut-points that we suggested in our submitted publication was directly coming from the observations we did in POPROW. It is a key feature in our future works and we will need to experiment on this enhancement in the next period.

The experiments run on WiSHFUL has so far lead to the submission of four papers, two of which have already been accepted; we plan the submission of an extended version of paper 1. to IEEE Transactions on Parallel and Distributed Systems:

1. "On the Distributed Computation of Load Centrality and Its Application to DV Routing" accepted at IEEE Infocom 2018, pre-print available at http://disi.unitn.it/locigno/preprints/Distributed_LoadC_Infocom2018.pdf
2. "Centrality-based Route Recovery in Wireless Mesh Networks", accepted at IEEE ICC 2018, pre-print available at http://disi.unitn.it/locigno/preprints/Centrality-Routing_ICC2018.pdf;
3. "RoRoute: Tools to Experiment with Routing Protocols in WMNs", accepted at IFIP/IEEE WONS 2018;
4. "Improving Routing Convergence with Centrality: Theory and Implementation of Pop-Routing", submitted to IEEE/ACM Transactions on Networking; this latter paper is the extension of the original Infocom paper presenting Pop-Routing, and POPROW contribution lies in the design and development of PRINCE.

We think that further publications may be submitted as the experiments on WiSHFUL already done are analyzed. Furthermore, extensions to journals of the conference papers submitted are possible depending on the feedback we get from the community. Finally, if more experiments can be run on the testbeds, it is possible (indeed highly probable) that more scientific results are published thanks to WiSHFUL and POPROW.

[1] The Guifi CN was recently awarded with the European Broadband Awards by the European Commission.

Coming to a more general impact, we are considering, after publishing the scientific results, to propose Pop-Routing for inclusion in standards (IETF, but also “de facto standards” including PRINCE as an option to widespread Open Source implementations of the routing protocols). This is however a long and resource consuming process, so its impact will be seen in years (not months) to come.

B.4.1 Value perceived

We have extensively used the test beds accessible thanks to the WiSHFUL project. The federation of different testbeds permits to seamlessly switch between them to run multiple experiments and obtain additional insights, as well as increasing the statistical confidence of the outcome. The WiSHFUL framework provides a set of functionalities that speed up the development of experiments. First of all, the controller-agent structure is extremely useful, as it makes it possible to write a single software program that controls all experiment nodes as if they were local. This hides the complexity of making remote procedure calls to the user. Second, the WiSHFUL UPIs permit to perform operations required for the experiments in a straightforward way. In addition, the framework is easy to extend, so a user can implement his/her own UPIs and use them for the purposes of the project.

B.4.2 Funding

Our experience with the testbed was in general positive, but in our specific case, we had to face several obstacles to reach our results, and we had to extend the duration of our experimental phase until the end of October, and indeed there are still many interesting experiments to be run.

Concerning the budget, we think it is a valuable and correct allocation, even if we dedicated to the project and experimental work more resources than covered by the budget itself. Our accounting tells that roughly 4PMs of senior researchers have been allocated to the project plus 6PMs of junior researchers, plus several master students that developed side software and extensions. One of the reasons, indeed the main one, is the fact that we discovered after the project begun that we could not use UPIs with OpenWRT, and we had to rewrite some of the instruments to prepare and run the tests. This required an amount of work we did not foresee at the beginning of the experiment. We think that this is in the normal course of action for academic research, so we don’t have regrets.

It must be noted, though, that the time needed to set-up fruitful experiments, extend the framework, develop scripts etc. is still quite long, as WiSHFUL is a research environment, so its tools still have a steep learning curve and several glitches and obstacles are often found on the path. Without the funding that was providing us the necessary labour resources and the necessary motivations, we would have not reached the breakthrough in the effort/results ratio.

Now that the project is over we would like to continue doing some experiments on the testbed in order to have more robust results, so we hope that access to the WiSHFUL resources can be granted on a voluntary, best-effort and gratuit base. We think that once WiSHFUL is mature, it is not unreasonable to pay a fee for it; however, it must be taken into account that academia is mostly non-profit, thus it is normally impossible to pay for an infrastructure without a project supporting it. We can imagine that for-profit organizations can pay fees, while academia can be involved, depending on the case, experiment and dedication, based on a free access (thus contributing only man power), or with with testbed extension, software development, and all maintenance effort that a large infrastructure requires.

Section C. Feedback to WiSHFUL

C.1 Testbeds/Hardware/Software Resources & UPIs used

C.1.1 Testbeds, Hardware and Software platforms

TESTBEDS	Used (Yes/No)
w.iLab.t (Heterogeneous wireless testbed @ iMinds, Ghent, Belgium)	Yes
IRIS (Software Defined Radio testbed @ TCD, Dublin, Ireland)	No
TWIST (Sensor testbed and openWRT router testbed @ TU Berlin, Berlin, Germany)	No
ORBIT (20 x 20 radio grid testbed @ Rutgers University, New Jersey, US)	No
FIBRE@UFRJ (OMF testbed @ UFRJ, Rio de Janeiro, Brazil)	No
WiSHFUL portable testbed	No

HARDWARE PLATFORMS			
Hardware	**Type**	**Technology**	**Used (Yes/No)**
wireless Wi-Fi card	Atheros athxk,	IEEE 802.11 a/b/g/n	Yes
	Broadcom b43	IEEE 802.11 b/g	No
Wireless sensor node	RM090	IEEE 802.15.4	No
	Zolertia Z1	IEEE 802.15.4	No
	Jennic JN516X	IEEE 802.15.4	No
Software Defined Radio (SDR)	WARPv3	IEEE 802.11 b/g	No
	USRP2-N210	2.4 – 2.5 GHz	No
		4.9 – 5.85 GHz	No
		50 – 860 MHz (RX only)	No
		800 – 1000 MHz	No
		1.5 – 2.1 GHz	No
		2.3 – 2.9 GHz	No
		50 MHz – 2.2 GHz	No
		400 MHz – 4.4 GHz	No

SOFTWARE	
OPERATING SYSTEMS	**Required (Yes/No)**
Linux	Yes
Contiki	No
TinyOS	No
PLATFORMS	**Required (Yes/No)**
Wireless MAC Processor (WMP)	No
Time-Annotated Instruction Set Computer (TAISC)	No
IRIS Software Radio	No
Generic Internet-of-Things ARchitecture (GITAR)	No

C.1.2 UPIs used

UPI Interfaces
Unified Programming Interface – Radio (UPI_R)
`set_tx_power()`
`set_modulation_rate()`
`interface_down()` (defined and implemented in this project)
Unified Programming Interface – Network (UPI_N)
`start_adhoc()`
`flush_iptables()` (defined and implemented in this project, used instead of clear_nf_tables())
`filter_mac()` (defined and implemented in this project)
`get_iface_hw_addr()`
Unified Programming Interface – Global (UPI_G)
`start_local_control_program()`
`get_hostname()` (defined and implemented in this project)
`run_terminal_command()` (defined and implemented in this project)

C.1.3 UPIs developed ad hoc

We briefly describe here the UPIs we proposed and we developed during this project :

- `interface_down()` : This UPI tears down a network interface. In our experiments, this is useful to simulate a node failure.
- `flush_iptables()` : This UPI deletes all rules from iptables. This could be done using the `clear_nf_tables()` UPI. The implementation of this UPI, however, uses the iptc Python library, so it does not work if the agent is not run with sudo privileges, which is not how the agents and the controller are started in the examples. So we decided to implement this UPI

which internally calls "`sudo iptables -F`". This is the same that is being done in the UPIs that manage WiFi interfaces.

- `filter_mac(mac_address)`: This UPI adds a DROP entry in the INPUT chain of iptables for the given MAC address, causing the node to discard all incoming packets from the specified address.
- `get_hostname()`: This UPI returns the hostname of the node, as this information is not yet included in the `Node` class used in WiSHFUL. By including the hostname in the WiSHFUL Node class, this UPI could be removed.
- `run_terminal_command(command, working_dir)`: This UPI runs a terminal command on the agent node. This is useful if we want to schedule a certain command at a specific time using the `exec_time()` modifier.

To extend and use our "ad hoc" UPIs we created some private copies of some of the WiSHFUL repositories on github (https://github.com/AdvancedNetworkingSystems/) and modified the setup configuration by editing the manifests repository. This way, the setup scripts install our modified version of the framework. These repositories include `wishful_upis`, `manifests`, `agent`, `controller`, `module_net_linux`, and `module_wifi`. If the consortium finds the extensions useful, they can be merged inside the main WiSHFUL repositories through pull requests.

C.2 Feedback on getting acquainted/using the testbeds offered in WiSHFUL

C.2.1 Feedback on testbed resources

Testbed usage went through two phases. The first one was getting acquainted with the testbeds and the frameworks. This phase only requires a limited number of nodes to test configuration scripts, node capabilities, and node setup. The initial experiments required only 2 to 4 nodes. We then started testing the configuration using the whole test bed, which means roughly 40 NUC nodes inside w-iLab.1. In this initial testing, we performed some very simple communication experiment trying to gain some insights on the actual network topology. We discovered that w-iLab.1 almost resembles a full-mesh, which is perfect for creating different topologies by disabling links selectively through MAC-layer filtering.

In the second phase we performed the experiments. In general, our experiments employed most nodes in the testbeds, as we tried to reproduce arbitrary topologies on top of a fully-meshed wireless network. For a single experiment shot we choose one topology, one node to kill, and we measure the performance of standard OLSR and of our Pop-Routing-enhanced version. A single shot lasts roughly 10 to 15 minutes, and we repeat each shot 10 times for statistical confidence. In addition, we choose multiple nodes to kill and different topologies. As an example, a single topology experiment with 5 nodes to kill and 10 repetitions lasts roughly 12 hours. Depending on the outcome, we might need additional runs to obtain additional insights or to fix some bugs, so an experiment session can last 2 to 3 days.

The usage/reservation ratio changed through the project. In the first phase the ratio was very low, as we were reserving the nodes to perform simple tests of our configuration scripts. In the second phase, instead, the ratio raised to almost 100%, as we were using all the reserved resources to perform our experiments. In this phase we executed roughly 800 experiment repetitions.

With respect to the originally planned usage, we had to give up using TP-Link routers in the TWIST testbed due to problems with node reservation. When reserving the whole TWIST testbed, TP-Link nodes sometimes failed to boot, blocking the reservation process. Sometimes we could figure out the faulty node, remove it from the slice, and try again reserving the resources, but this did not always work. We notified these problems to TWIST maintainers, which tried to help us as much as possible

with testbed issues. However, as time was running out, we preferred to focus our effort in the w.iLab1 testbed, as it is only composed by NUC nodes and gave use the highest flexibility. We however plan to continue our analysis using the TWIST testbed, as we need to understand whether the limited computational resources of OpenWRT nodes are enough to support the additional computational burden introduced by Prince.

C.2.2 Feedback on experimentation tools

Tools	**Used? (YES/NO)**	**Experience with the tool. What were the positive aspects? What didn't work?**
Fed4FIRE portal	No	
Testbed specific portal	Yes	The testbed portals give important information on the testbed setup. For example, the TWIST Wireless testbed portal includes detailed instructions on how to setup the experiments, controlling them, and accessing the nodes. In addition, it provides a very clear map of the testbed, permitting to choose the nodes depending on your needs.
jFed	Yes	jFed is very useful at the beginning when trying to setup the first test experiments with a few nodes. However, reserving a full testbed requires a lot of manual setup. At the beginning, we generated an .rspec file including all testbed nodes, but when starting the experiment jFed does not check for actual availability. The only way is trying to start the experiment and, in case of failure, manually remove the nodes that are not available from the experiment. Due to this, we switched to OMNI.
OMF	No	
OML	No	
SSH	Yes	SSH is fundamental for accessing and controlling the nodes. The private/public key authentication mechanism is very handy.
Ansible[2]	Yes	Ansible was used to issue commands to the nodes using SSH as the "low-level" mean. It has been incredibly useful for setting up the nodes and configuring them. We faced some problems with the w.iLab.1 testbed, as we had been blocked due to (we believe) too many parallel SSH connections. We do not know if this is an SSH server configuration or a firewall problem, but to overcome this we had to run ansible on one of the testbed nodes, instead of using a desktop PC in our lab.
OMNI[4]	Yes	Due to the low user friendliness of jFed with respect to the reservation of large number of nodes, we used OMNI to query the status of the nodes of a chosen testbed and, based on the answer, generate an .rspec file including all (or a chosen subset of) the available nodes. In addition, we use OMNI to reserve the nodes using the generated .rspec file. We designed the scripts in cooperation with TU Berlin, starting from some scripts provided by them. If considered useful, we plan to release the scripts to the community.

[2] Added by POPROW Team.

C.2.3 Feedback on getting acquainted with experimentation

We started learning how to use the testbed by following the instruction on the TWIST wireless testbed page[3]. The procedure to obtain an account, a certificate, and to setup jFed is well-written. The description, in our opinion, lacks some information regarding the setup of the nodes. It was not clear to us how to install the software required by our experiments (e.g., OLSR daemon). With the help of TU Berlin we understood that we are given root access to the devices and we can thus install any required software at our will. We believe that a simple sentence on the website could help future users.

In addition, we believed that testbed nodes would be "WiSHFUL-ready" meaning that, after booting, the WiSHFUL framework would already be installed. Instead, we learned that the framework must be installed by the users during the setup phase. This definitely makes sense, as the user might modify the framework and install his/her own version of the framework. As a "beginner", however, the reason was not immediately clear.

Finally, the fact that the framework is only currently supported on Debian-like systems prevented us from performing experiments on TWIST, as the TWIST testbed is mainly composed of OpenWRT nodes. As the TWIST test-bed was in the Open Call, we assumed that OpenWRT was supported. It might be useful to mention that the WiSHFUL framework is only currently supported on the NUC nodes of the test bed. In addition, porting the framework to OpenWRT would make it even more appealing for research projects.

C.2.4 Feedback on using the testbeds

The planned project time of 6 months was just enough to obtain some meaningful results. The problem was not only the initial setup phase, but also understanding problems related to the physical topology and the software we were using. Indeed, to be completely honest, we think that for many complex experiments 6 months are not really enough, so a flexible framework of time allocation going from 4 to 9 months depending on the experimental complexity would be better.

With respect to the physical topology, we had to deal with the fact that, even within the w.iLab1 full-mesh, not all links were equal. This was a problem when "implementing" our logical topology on top of the physical one, as link qualities were time-varying. This causes the shortest paths to change from experiment to experiment, making it difficult to obtain consistent and comparable results. To deal with this problem, we had to introduce random link cost multipliers in the OLSR configuration file. This, however, simply reduces the probability of this problem to occur, but does not totally solves it. We are aware that "route flapping" is an existing phenomenon in real networks, thus it is normal to encounter it also in test beds; however, packing tens of nodes within communication range exacerbate it.

Even if it is not only related to the test bed, but to the entire experimental environment we set up and used, we'd like to comment also on some software and parameters setting at large. Experimenting with OLSR parameters, we found some that were completely changing the outcome of the experiments. One example is the validity time of Hello and TC messages. For example, setting a Hello timer value of 2 s and a Hello time validity of 6 s permits to be very reactive to link losses. However, given the physical topology, the level of interference caused by using all nodes together causes the probability of losing three packets in a row non-negligible. This causes routes to change for reasons that are not under our control, polluting the results. An additional example is the "FishEye" routing algorithm embedded in OLSR, which tries to reduce the overhead of the protocol due to TC message flooding by using a sequence of TTL values. For example, by using a TTL sequence

[3] https://www.twist.tu-berlin.de/tutorials/twist-am-usage.html

of 1, 2, 4, 8, one-hop neighbors receive TC messages of a node every TC interval, 2 hop neighbors every 2 TC intervals, 3 and 4 hop neighbors receive TC messages every 3 TC intervals, while 5, 6, 7, and 8 hop-neighbors receive TC messages every 4 TC intervals. This mechanism, combined with the TC timer validity, can cause links to be invalidated even though they are not, so we had to manually disable it in our experiments.

On the positive side of the experience, instead, we gained a lot of insights about the working principles of OLSR in a real environment, and this was only possible thanks to this project. We thus plan to continue using the testbed for future projects, as the knowledge gained by performing real world experiments is priceless. In addition, we also plan additional Pop-Routing experiments for a more in-depth evaluation.

C.3 Feedback on using UPIs and extending the WiSHFUL framework

C.3.1 Getting acquainted

The "HOW-TOs" for reserving the nodes, starting an experiment, and getting access to the nodes was very clear, especially on the TWIST website. With very few help from TU-Berlin we were able to start straight away.

As previously mentioned, however, it was not immediately clear to us that testbed nodes are not WiSHFUL-preconfigured and that the framework is only currently only available for Debian-like systems.

C.3.2 Using (and Extending) the WiSHFUL framework

We believe that the WiSHFUL framework is a big step towards seamless control of network devices for experimentation. Configuring and controlling network interfaces with a standard interface can be very helpful for research activities, as it would remove a large part of the testbed setup overhead.

The bootstrap phase was, however, not easy. Understanding how to properly use the control-agent architecture and how to send commands to agent nodes was not obvious. We believe a simple introduction tutorial would be very useful for the community: an example with a few line of codes where a controller waits for a remote agent and, when ready, invokes one or two UPIs on that agent as an example. This, coupled with a webpage describing the working concept and the few lines of codes would already provide the basis for all users. Once we understood the basic concepts, indeed, it was very easy for us to implement complex features in our controller. It is true that on the WiSHFUL website we can find some "getting started" videos, but they are not that "handy" to start with, as they are very long.

The initial setup was not complete, e.g., some Python libraries required for our experiments were not installed. This is clearly not an issue, as users have the freedom of installing any required library/software. The fact that users have complete freedom in the setup of the nodes is actually a plus.

The technical offering, from a functionality point of view, is still in its development phase, which is more than reasonable. Some UPIs required by our experiments do not exist or they do not work as expected. After understanding the UPI working principle, however, we were able to quickly define and implement our custom UPIs. This means that if a specific UPI does not exist, the WiSFHUL framework permits any user to create its own exploiting the core of the framework. If every project would share their extensions, as we plan to do, the number of UPIs would increase rapidly, making it a complete and solid framework. For this to happen, however, the software design principles used in the implementation should be clearly defined beforehand. As an example, we have found some UPIs using root terminal commands for executing tasks (e.g., "`sudo iwconfig ...`"), while others

using Python libraries (e.g., "`iptc`" for IPTables management). UPIs using sudo commands work even if the process is launched by a normal user (having sudo privileges), while the usage of Python libraries requires the agent to be started with the root user. Which one of the two choices is the official one is not clearly defined. Having the code base consistently written would help in this regard.

In conclusion, we believe that WiSHFUL offers very powerful tools for experimentation. Especially because of its easy extensibility, we will definitely use it in the future to continue our experiments and to carry out new ones.

C.4 Feedback on the administration process and support received

The administration of WiSHFUL is lightweight and efficient, we experienced (so far) no issues or problems. We find the form of contract as third parties instead of being included in the consortium definitely a good choice, simplifying greatly not only management, but also the initial setup signing the contract and finding the proper timeframe to start the project and run the experiments. In addition, given the limited budget of these projects, it greatly reduces the overhead for the participants, leaving more resources for technical and scientific work.

The communication with the Patron and the Consortium in general has been effective and timely. We regret that we were unable in the end to use also the TWIST testbed due to lack of support on OpenWRT, as well as construction works that prevented the testbed to have a stable configuration in time, so that results of the experiments would have been meaningless and not reproducible. However, we are aware of the difficulties and real-life limitations and we are very satisfied with the use of w.iLab1 in the end.

C.5 Why WiSHFUL was useful?

Concerning WiSHFUL testbeds, we find particularly convenient the possibility of reserving them through a common tool and having a full root access to the nodes, permitting researchers to perform experiments in complete freedom.

The WiSHFUL framework lightens the burden of controlling experiments and the nodes of a test bed. Given the solid base framework, we have found it extremely useful even if all required functionalities were not implemented, as it was easy to extend it to meet our requirements.

Added value of WiSHFUL

We already commented throughout the document how being able to systematically perform experiments in a real testbed was a key to the advancement of our research. The comments on Figure 19 are a good summary of this, showing how repeated experiments in a controlled environment are fundamental to gain proper insight into results.

What is missing from your perspective?

We would have greatly appreciated support for the OpenWRT platform, as this is the de facto standard for wireless networking in the open source arena. Most of the real routers run OpenWRT and it has support for all the most popular protocols.

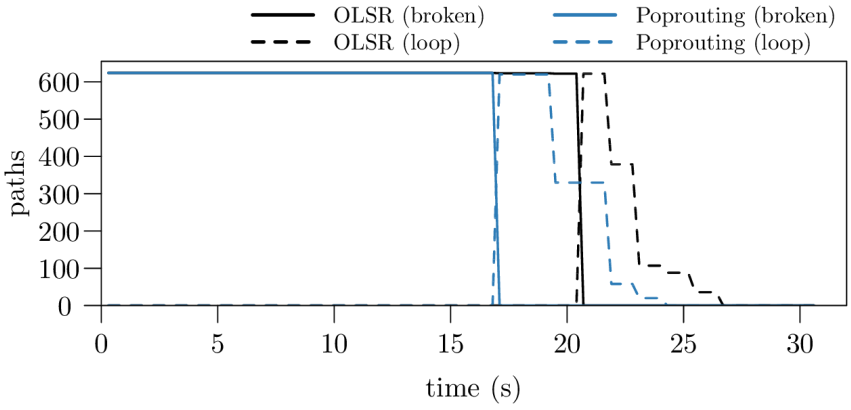
In addition, a basic online tutorial with a simple controller-agent experiment using a couple of UPIs would be very useful to start understanding the working principles of the framework. In the repository there are many examples, but the documentation is lacking, and some of them are too complicated to start with.

C.7 A possible quote

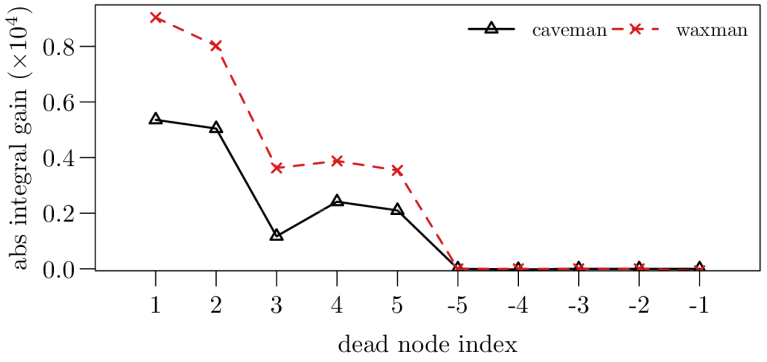
The step from emulation/simulation to a real-world solution is extremely challenging and requires resources. WiSHFUL was important for our research because it made it possible for us to do a key step in this direction sparing us the tremendous effort needed to set-up and control a real test bed.

Section D. Leaflet

- Title: POPROW: Pop-Routing on WiSHFUL
- Advanced Networking Systems lab, Department of Computer Science and Information Engineering, University of Trento: http://ans.disi.unitn.it.
- Goals: The goal of POPROW is to test and enhance “Pop-Routing”, a technique for wireless mesh link-state routing protocols that tunes the generation frequency of control messages independently for each node as the result of real-time graph analysis performed on the network topology. Pop-Routing makes it possible to reduce the routing tables convergence time after a failure, and represents a novel approach to guarantee network scalability in mesh networks.
- Main Challenges: To deploy Pop-Routing in a real network environment after it was previously tested in emulated environment. To verify the effectiveness of Pop-Routing in a context that presents all the complex interaction of a real network, and are hidden in an emulated one. To stabilize the code in order to make it suitable for production networks.
- Main Results:



The convergence time of shortest paths for OLSR and Pop-Routing upon node failure.



The Gain in convergence time averaged on multiple failures on the several nodes.

- Conclusions: WiSHFUL has been essential to make the code-base of Pop-Routing more stable, spot several bugs that were present in the code and make it close to production-state. It also raised several issues and opened new research directions that would have been otherwise

not possible to foresee. The results are at the base of three publications submitted and under revision at the end of the project.

- Feedback: The step from emulation/simulation to a real-world solution is extremely challenging and requires resources. WiSHFUL was important for our research because it made it possible for us to do a key step in this direction sparing us the tremendous effort needed to set-up a real testbed.