On the Distributed Computation of Load Centrality and Its Application to DV Routing

Leonardo Maccari*, Lorenzo Ghiro*, Alessio Guerrieri†, Alberto Montresor*, Renato Lo Cigno*

*DISI, University of Trento
†SpazioDati srl

***Abstract*—Centrality metrics are a key instrument for graph analysis and play a central role in many problems related to networking such as service placement, robustness analysis and network optimization. Betweenness centrality is one of the most popular and well-studied metric. While distributed algorithms to compute this metric exist, they are either approximated or limited to certain topologies (directed acyclic graphs or trees). Exact distributed algorithms for betweenness centrality are computationally complex, because its calculation requires the knowledge of all possible shortest paths within the graph. In this paper we consider *load centrality*, a metric that usually converges to betweenness, and we present the first distributed and exact algorithm to compute it. We prove its convergence, we estimate its complexity and we show it is directly applicable–with minimal modifications–to any distance-vector routing protocol based on Bellman-Ford. We finally implement it on top of the Babel routing protocol and we show that, exploiting centrality, we can significantly reduce Babel's convergence time upon node failure without increasing signalling overhead.**

Our contribution is relevant in the realm of wireless distributed networks, but the algorithm can be adopted in any distributed system where it is not possible, or computationally impractical, to reconstruct the whole network graph at each node and compute betweenness centrality with the classical approach based on Dijkstra's algorithm.

***Index Terms*—Multi-hop networks; Mesh networks; Ad-hoc networks; Bellman-Ford; Load centrality; Distributed Algorithms; Failure recovery**

I. Introduction

A centrality indicator measures the prominence of a vertex with respect to the graph structure, based on a specific metric. A vertex with high centrality, depending on the metric definition, is either one with many neighbors, or with low average distance to all other vertices, or one that controls many flows between other vertices, and so forth. Centrality is relevant in many sciences interested in network analysis. It was initially introduced by social scientists to identify influential people in social networks, but it has become a common tool in computer and communication networks too [1], [2].

Among the several centrality indexes proposed in the literature, a noteworthy one is *Betweenness Centrality* (BC): it measures, for each vertex, the fraction of global shortest paths passing through that vertex [3]. In computer networks, BC has been used for service placement [4], to improve routing [5], [6], for topology control [7], [8], security [9], [10], and for several other uses [2], [11]. The application of load centrality in this paper is toward improving scalability and resilience of routing protocols.

BC requires information about all shortest paths between any pair of vertices, so it implies a full network graph knowledge and is generally computed off-line. Such centralized implementations do not integrate well with distributed protocols; this observation alone hampers the adoption of centrality metrics to improve, for instance, distance-vector (DV) routing protocols.

Distributed, approximated algorithms are available for generic graphs [12], [13] and exact ones only for selected topologies. To the best of our knowledge there is currently no distributed algorithm able to exactly compute BC on a generic network graph without relying on the knowledge of the complete topology.

In this paper we consider a variant of BC, called *Load Centrality* (LC) [3], [14], that lends itself to be computed in a distributed way. We present an efficient distributed algorithm to calculate it defined on top of Bellman-Ford. It is thus directly applicable to any DV routing protocol with minimal modifications. Our algorithm computes the exact LC, which estimates the expected load on a vertex contemplating multipath routing on paths with equal weight. In the simpler case, with only one minimum-weight path per pair of vertices, LC converges to BC (see Sec. III). Still, our algorithm can be applied to any kind of weighted and unweighted graphs. Its convergence time grows linearly with the network diameter D: after a time proportional to at most $3 \times D$, every node knows its own centrality and the centrality of all the other nodes.

In light of this discussion, our core contributions are presented in the four sections of this paper: Sec. IV introduces the algorithm and gives a proof of its convergence, Sec. V shows how to integrate it into Bellman-Ford; Sec. VI presents the convergence properties using synthetic topologies with controlled properties; Sec. VII documents our extension of `babeld`, the open-source implementation of the widely used DV routing protocol for mesh networks specified in RFC 6126 [15]. It also shows that, exploiting the notion of LC, the protocol convergence time can be improved up to 13% without increasing signalling overhead.

This work has been partially funded by the European Commission, H2020-ICT-2015 Programme, Grant Number 688768 'netCommons' (Network Infrastructure as Commons) and the H2020 GA No. 645274 "Wireless Software and Hardware platforms for Flexible and Unified radio and network controL (WiISHFUL)" with the project "Pop-Routing On WiSHFUL (POPROW)" financed in Open Call 3.

The paper is completed by a state-of-the-art analysis (Sec. II) and by the problem definition (Sec. III); Sec. VIII concludes the paper.

II. State of the Art

Centrality measures have been used to enhance traffic monitoring [5], [9], intrusion detection [16], resource allocation [2] and topology control [7].

Among all, *Betweenness Centrality* (BC) is a remarkable centrality index which is computed, for all nodes of a network, with the well-known centralized Brandes' algorithm [17]. It executes an instance of Dijkstra's algorithm rooted on each vertex of the graph and in parallel it updates the BC indexes. In a network with n nodes and m weighted edges, the computational complexity of this approach is $O(nm + n^2 \log n)$. Brandes' algorithm can be adapted to compute other centrality indexes based on minimum-weight paths [3]. For example, Dolev et al. proposed a generalization of BC to deal with different routing policies [5].

There are two main problems that hinder the use of centralized algorithms in a network of routers. First of all, only link-state (LS) protocols provide information on the whole network topology to nodes; a mandatory requirement to perform the centralized BC calculation. DV protocols are completely excluded. Secondly, even in LS protocols, as the network size becomes increasingly large, centrality metrics may require excessive computational resources. Despite the introduction of several heuristics [18], the online computation of the indexes on low-power hardware requires several seconds and is generally not possible in real-time on large networks [10].

One natural approach to speed-up the computation is random sampling [12], [13], [19]–[22]. Independently from each other, Jacob et al. [12] and Brandes and Pich [13] proposed approximated algorithms that only consider contributions from a subset of vertices sampled uniformly at random. Later proposals [23], [24] can compute BC with adjustable accuracy and confidence. More recently, dynamism has been taken into consideration, with several algorithms able to update BC on evolving graphs [25]–[28].

These randomized algorithms are fast, but still centralized. Distributed algorithms for the exact computation of centrality with sufficiently good scalability properties have been proposed, based on a dynamical system approach, but only for specific topologies (DAGs and trees) [29]–[31].

The LC metric is similar to BC, and sometimes it is confused and mistaken for it [3]; indeed, they converge to the same metric when there is a single minimum-weight path between any couple of nodes. To the best of our knowledge, this paper provides an algorithm for the exact distributed computation of LC for the first time.

III. Definitions

Let $G(\mathcal{V}, \mathcal{E})$ be a graph where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. Our contribution is not limited to communication networks, so we provide definitions as general as possible. To clarify the context we use the terms vertices/edges when referring to a generic graph, and nodes/links when referring to a communication network.

Load Centrality is defined as follows [3]:

Definition 1 (Load Centrality (LC))**.** *Consider a graph $G(\mathcal{V}, \mathcal{E})$ and an algorithm to define the (potentially multiple) minimum weight path(s) between any pair of vertices (s, d). Let $\theta_{s,d}$ be a quantity of a commodity that is sent from vertex s to vertex d. We assume the commodity is always passed to the next hop following the minimum weight paths, and in case of more than one next hop, traffic is divided equally among them. We call $\theta_{s,d}(v)$ the overall commodity forwarded by vertex v. The Load Centrality of v is given by:*

$$LC(v) = \sum_{s,d \in \mathcal{V}} \theta_{s,d}(v)$$

Normally it is assumed that $s \neq d$, $s \neq v$, $d \neq v$ and in general also $\theta_{s,d} = 1$. The latter makes LC a property fully defined by the graph structure and by the algorithm used to discover minimum weight paths. In that case, if the graph is undirected there are $\frac{N(N-1)}{2}$ couples (s, d) and LC can be normalized as

$$\overline{LC}(v) = \frac{2}{N(N-1)} \sum_{s,d \in \mathcal{V}} \theta_{s,d}(v)$$

BC is instead formally defined (see Freeman [32]) as follows:

Definition 2 (Betweenness Centrality (BC))**.** *We call σ_{sd} the number of minimum weight paths between vertex s and vertex d, and we call $\sigma_{sd}(v)$ the number of those minimum weight paths passing through vertex v. Betweenness Centrality is defined as:*

$$BC(v) = \frac{2}{N(N-1)} \sum_{s,d \in \mathcal{V}} \frac{\sigma_{sd}(v)}{\sigma_{sd}} \tag{1}$$

Again, normally it is assumed that $s \neq d$, $s \neq v$, $d \neq v$.

LC and BC are very similar, but they do not coincide. Consider Fig. 1, reporting a sample network annotated with the values of LC and BC on each node, assuming every edge has the same weight. If the commodity moving from s to d is split equally between two next hops that lie on two paths with equivalent total weight, intuitively node v and w will both carry half of it. This is what LC measures. BC, instead,

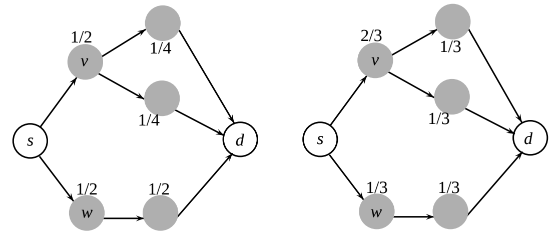


Figure 1. Difference between values of normalized load (left) and betweenness (right) centrality in the same network, taken from [3].

reflects that at the right side of node v there are more minimum weight paths towards d than at the right side of node w. Since BC counts the fraction of minimum weight paths, v turns out to be more central than w.

Note that, in communication networks, if centrality is used to estimate IP traffic relayed by a node (bit/s, number of open TCP connections, active HTTP sessions ...) LC may give a more accurate estimation than BC. If the IP routing protocol does not support some kind of multipath routing or load balancing, one minimum weight path is used and traffic is never split. In this case, the normalized LC coincides with the BC, but the modern trend to use multiple paths as in SCTP (Stream Control Transport Protocol) or Multipath TCP makes LC an interesting metric for traffic management.

IV. Distributed Load Centrality Computation

In order to compute LC for each network's vertex, we propose a distributed algorithm that can be easily integrated into any routing protocol that keeps a routing table up-to-date. The algorithm, as executed by vertex v, is shown in Algorithm 1, while Tab. I lists all information maintained by v. Recall that having an updated routing table is in general not sufficient to compute centrality metrics, since they mandate a full topological knowledge, not only the next hop toward a destination.

Algorithm 1 is based on the commodity diffusion process described in the definition of LC (see Definition 1). Each vertex generates a unitary amount of commodity for all possible destinations; such commodity is split and aggregated along the route to destinations.

The routing protocol keeps an up-to-date list of next hops in vector NH, where $NH[d]$ are the next hops to reach destination d. Our algorithm computes the complementary vector PH, where $PH[d]$ are the *previous hops* from which the commodity going toward d is coming. This is obtained by periodically sending a message $\langle v, NH, contrib\rangle$ to all neighbors of v; when these messages are received by each next hop, PH is updated. The previous hops stored in $PH[d]$ are used to aggregate all the incoming commodity toward d before splitting it among all next hops.

Table I
Variables used by each vertex v in Algorithm 1

Symbol	Description
$\mathcal{V}$	The set of all vertices
$neighbors$	The set of neighbors of v
NH	For each destination $d \neq v$, the vector $NH[d]$ is the set of vertices used by v as next hops to reach d
PH	For each destination $d \neq v$, vector $PH[d]$ is the set of *previous hops*, i.e., vertices that list v as one of the next hops to reach d
$loadOut$	For each destination $d \neq v$, $loadOut[d]$ is the overall commodity passing through v to reach d
$contrib$	For each destination $d \neq v$, $contrib[d]$ is the contribution that v will send to each of its next hops to reach d, equal to $loadOut[d]/\lvert NH[d]\rvert$
$loadIn^u$	For each neighbor u and for each destination $d \neq v$, $loadIn^u[d]$ is the commodity's contribution that vertex u sends to v toward d (as reported by u to v)
$load$	The approximation of load centrality known so far

Alg. 1: General distributed Protocol (executed by v)

```
1  Init:
2      foreach d ∈ V − {v} do
3          foreach u ∈ neighbors do
4              loadIn^u[d] = 0;
5          PH[d] = [ ];

6  Repeat every δ s:
7      load = 0;
8      loadOut[v] = contrib[v] = 0;
9      foreach d ∈ V − {v} do
10         loadOut[d] = 1 + Σ_{u∈PH[d]} loadIn^u[d];
11         contrib[d] = loadOut[d]/|NH[d]|;
12         load = load + loadOut[d];
13     send ⟨v, NH, contrib⟩ to neighbors;

14 on receive ⟨u, NH^u, contrib^u⟩ from u do
15     foreach d ∈ V − {v} do
16         if v ∈ NH^u[d] then
17             PH[d].add(u);
18             loadIn^u[d] = contrib^u[d];
19         else
20             PH[d].delete(u);
```

The rest of Algorithm 1 is designed to maintain information about incoming and outgoing commodity. In particular, dictionary $loadOut$ stores the overall commodity passing through v to reach every possible destination, while $contrib$ stores the commodity's contributions that v sends to each of its next hops. Note that having both $loadOut$ and $contrib$ is redundant; $loadOut$ is introduced only to clarify the algorithm and simplify the proof that $load$ converges to LC.

During initialization (line 1-5), the commodity coming from every neighbour is set to 0, while waiting for more up-to-date information to come. PH entries are initialized to the empty vector as well.

Periodically, each vertex v re-computes (for every destination d) its contribution to $load$ for its next hops and sends this contribution to all its neighbors with the message $\langle v, NH, contrib\rangle$ (line 6-13). The contribution is given by 1 (its unit contribution to the load addressed to d) plus all contributions received so far, divided among all vertices which are next hops for destination d (computed on line 10-12).

Whenever a message is received from vertex u, vertex v first updates the previous hop set PH, by either adding (line 17) or deleting (line 20) the vertex u. Then, it copies the contributions toward every d computed by u and received in the message $\langle u, NH^u, contrib^u\rangle$ into $loadIn^u$ (line 18).

We show now that at steady state, under sufficiently stable conditions, $load$ in Algorithm 1 converges to the correct LC for each vertex.

Theorem 1. *Let $G = \{G_d = (\mathcal{V}, E_d) : d \in \mathcal{V}\}$ be the collection of all routing graphs induced by all nodes running* *an underlying routing protocol:*

$$E_d = \{(i,j) : i \in NH_j\}$$

If G remains stable for a long enough period of time then, for each node v, the 'load' variable maintained by v will eventually converge to the correct LC for v.

Proof. Given a node v, we prove that for each destination d, the commodity that v forwards toward d is eventually computed in a correct way. Since the overall commodity forwarded by v towards any possible destinations is periodically aggregated into variable $load$, this proves the theorem.

For each destination d, the routing protocol generates a routing graph: a loop-free directed acyclic graph (DAG) made of all the (potentially multiple) minimum weight paths ending in d. Let $S = \{s = u_0, u_1, u_2, \ldots, u_{|\mathcal{V}|} = d\}$ be a sequence representing a topological sort of the DAG G_d. We prove that each node in the sequence correctly computes the load that is passing through v, by induction on the sequence of nodes.

In the first node u_0, $PH[d]$ is empty (because G_d is a DAG). Thus, $loadOut[d]$ is set to 1, which is the correct value for the load passing through this node. This load is then divided equally among all nodes in $NH[d]$.

Now, consider node u_k and assume all preceding nodes in the sequence have already computed the correct value for their variable $loadOut[d]$. Each node $u \in PH[d]$ is included in $\{u_0 \ldots u_{k-1}\}$, thanks to the topological sort. Thus, eventually all of them will send a message to v, updating the corresponding entries in variable $loadIn$.

As soon as node v receives all the necessary information from all nodes in $PH[d]$, variable $loadOut[d]$ will contain the correct value.

A special case is given by node d, where $loadOut[d] = 0$. □

The theorem assumes minimum weight paths are stable long enough to allow the centrality computation to converge. In case of dynamism, results can be temporarily different from the correct ones, until routing paths stabilize again. At that point, given all the needed information is periodically broadcast to all vertices, Algorithm 1 converges again to the correct LC values.

We can estimate the convergence time of Algorithm 1 in the worst case scenario. We assume all clocks are synchronized and time required to propagate data along an edge is very small compared to δ, but not null. Therefore, vertex v always receives updates from neighbors after it sent its own updates, and time needed to propagate information on x hops is always $x \times \delta$s. We also assume that our algorithm starts after the routing protocol convergence. Under this assumption, the following corollary holds:

Corollary 1. *Given a graph G with diameter D, the convergence time Δ_t of our algorithm is in the worst case proportional to $D - 1$.*

Proof. A vertex v converges when its $load$ sums all contributions from all minimum weight paths crossing v. Consider the load on v generated from s and directed to d for which v is in at least one of the minimum weight paths. If $v = s$ or $v = d$, convergence is immediate, as no contribution will be received. Let S be the graph ordering relative to G_d, and $v = u_k$. If $k = 1$ then v converges after receiving the contribution from s, that is, after δs. Otherwise v converges when the load is propagated from s to v, which requires $k \times \delta$ intervals. In the worst case $v = u_{D-1}$ and the $load$ of v converges in $\delta(D-1)$s. □

Generally, at v the own centrality value is useful only if compared to other's centrality. This is why after the complete convergence of centrality in the network another message could be sent (and forwarded) by each vertex carrying its own value of centrality. In conclusion, always in the worst case scenario and after routing table convergence, the time required to perform centrality computation and dissemination will be proportional approximately to $2 \times D$.

So far we described how to compute LC with a distributed algorithm on an arbitrary topology when an abstract routing protocol is available, thus the convergence proof and also the convergence time are fully general. The next section documents the implementation of Algorithm 1 integrated with a popular DV routing protocol, instead of applying it after the routing has converged.

V. Implementation on top of DV routing

A DV routing protocol (like the classical RIP, or Cisco EIGRP, or the Babel protocol adopted in this work) uses the Bellman-Ford algorithm to maintain a Routing Table (RT) mapping each destination to a next hop and a distance. We extended the RT as shown in Tab. II, in which d and m are the typical entries of any RT, and the other data structures associated to d are added by our algorithm[1]. Note that NH is an array of next hops, therefore we support routing protocols which implement multipath routing.

Table II
Extended Routing Table

Field	Notation	Description
Destination	d	the destination identifier d
Metric	m	the distance of v from d according to a given metric
Next-hop list	NH	a list of next hops NH
Incoming Load	$loadIn$	a dictionary mapping a neighbour u in PH to its contribution c for destination d
Load	$load$	the load centrality of d, as far as v knows

The computation of LC together with DV routing works as described in Algorithm 2. At start-up, the RT is initialized, we store the load for node v running the protocol in its own

[1]We use square brackets operator to represent access to a tuple related to destination d, i.e., $RT[d] = (m, NH, loadIn, load)$. We use the dotted notation to access single fields of the tuple, as in $RT[d].load$. For a dictionary, the '()' operator returns the list of the keys of the dictionary, i.e., $RT() = [d_0, d_1 \ldots d_N]$. The '[]' and '{}' symbols refer to the empty list and dictionary, respectively.

RT and we set it to zero (line 1-2). Line 5-10 compute the distance vector and send it to neighbors. The vector contains the destination and the metric plus all available next hops (in case there are multiple equivalent paths to destination). It also includes the load sent by v toward the destination, aggregated in line 7-9, and the current estimate of destination's load as far as v knows. Here we use the distance vector also to propagate the supposed load of each node, including v itself, without requiring another packet after convergence. Note that the "send" action in line 10 is generally batched in order to send one single message containing the whole vector. Line 12-16 are the core of Bellman-Ford algorithm, with multipath routing (line 15-16). Note that we assume there is another process performing link sensing, for instance with neighbors exchanging periodic HELLO messages. This is fundamental in any routing protocol and keeps updated a dictionary of neighbors with respective link costs $C[\cdot]$. Line 17-20 computes the contribution of u towards d, dividing it by the size of the next hop array, to take multipath into account. Line 21-22 updates load of d if the value is proposed by the neighbour chosen as next hop towards d. Finally, on line 23-26, v computes its own value of load, which will be propagated with the next distance vector.

Compared to Algorithm 1, in Algorithm 2 we do not wait DV convergence to start computing centrality; instead, the two processes run concurrently (but do not interfere, so both converge). This, together with the removal of perfect synchronization introduced in the theoretical analysis, improves average convergence properties of the algorithm. The next section studies the convergence time of Algorithm 2 with simulation experiments.

Alg. 2: Integration with DV protocol (executed by v)

```
1  Init:
2      RT[v].m = RT[v].load = 0;
3      RT[v].NH = [ ];
4      RT[v].loadIn = { };

5  Repeat every δ s:
6      foreach d ∈ RT() do
7          loadOut = 1;
8          foreach u ∈ RT[d].loadIn() do
9              loadOut += RT[d].loadIn[u];
10         send ⟨d, RT[d].m, RT[d].NH, loadOut, RT[d].load⟩
             to neighbors;

11 on receive ⟨d, m, NH^u, loadOut^u, load⟩ from u do
       /* Bellman-Ford */
12     if d ∉ RT() OR m + C[u] < RT[d].m then
13         RT[d].NH = [u];
14         RT[d].m = m + C[u];
15     else if m + C[u] == RT[d].m then
16         RT[d].NH.append(u);
       /* Manage load contributes */
17     if v ∈ NH^u then
18         RT[d].loadIn[u] = loadOut^u/|NH^u|;
19     else
20         RT[d].loadIn.remove(u);
       /* Load indexes propagation */
21     if u ∈ RT[d].NH then
22         RT[d].load = load;
       /* Own load update */
23     RT[v].load = 0;
24     foreach d ∈ RT() − {v} do
25         foreach u ∈ RT[d].loadIn() do
26             RT[v] += RT[d].loadIn[u];

/* Dictionary C[·] contains the cost of the links to neighbors */
```

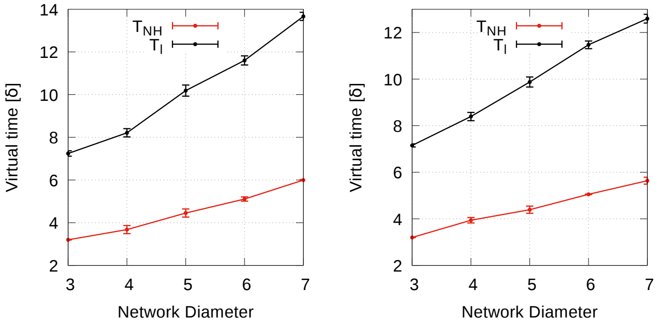
VI. Convergence Study

We developed a Python simulator which takes as input a network graph and implements all the send and receive actions of Algorithm 2. The simulator uses a virtual clock and triggers a send-event once per virtual time unit for each node, thus $\delta = 1$, and we can refer to convergence time simply as multiples of δ. A small random jitter is added to events scheduled by nodes to avoid perfect synchronization.

A simulation ends when all nodes converge to steady state, i.e., when the RT has one line per destination and the NH and the $load$ fields do not change anymore. We separately measure the time needed for full convergence of NH (T_{NH}) and $load$ (T_l). Full convergence is determined by the slowest node convergence time. Since we demonstrated that convergence time should grow, in the worst case, linearly with the network diameter D, we generated graphs with growing $D \in \{3, 4, 5, 6, 7\}$. For each D, we tested two different graph models: The classical Erdős and the Barabási-Albert model. For both we averaged results over 40 different graphs with 1000 nodes each.

Figs. 2a and 2b report the average (with 99% confidence intervals) of T_{NH} and T_l simulated over Barabási-Albert and Erdős graphs respectively. Both figures show how the growth of convergence time is approximately linear with the diameter growth, confirming the theoretical result of Sec. IV. Note also that it takes approximately D time units to let the RT converge, which is expected, because the distance vector must be propagated from every node to every other node, so T_{NH} depends on the network diameter. Since we add some jitter, generation of distance vectors is not synchronized, thus it may be that node i receives an update from a neighbor j before generating its own. In this case, in the same time unit, the information is sent from j to i and propagated from i to its neighbors, so it can take less than D time units for information to travel on the longest path. This explains why in the figures T_{NH} is generally smaller than D.

After convergence of NH, our theoretical analysis predicts a time of $2 \times (D - 1)$ to achieve full load convergence, while simulations show that it requires much less than that. This improvement is given by the parallelization of the two processes and can be seen in Fig. 3, which reports all values of T_{NH} and T_l, together with another value T_{sl} which indicates the time at which each node converges to its own value of centrality. We call this value *self convergence time*, as opposed



(a) Barabási-Albert networks. (b) Erdős networks.

Figure 2. T_{NH} and T_l Vs network diameter, with 99% confidence interval,for Barabási-Albert and Erdős networks.

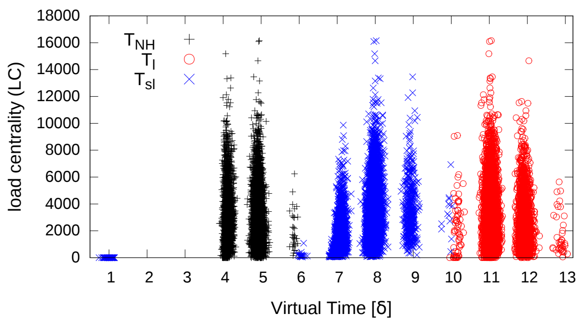


Figure 3. T_{NH}, T_l, T_{sl} for all the nodes, 40 simulations, Erdős graphs with diameter 7.

to the full convergence time defined before.

Fig. 3 was computed for all the 40 Erdős graphs with diameter 7 (networks with different parameters are not shown as they behave similarly) and shows that at time 6, when the last RT converges, some nodes already reach self convergence. Similarly at time 10, when all nodes reached self convergence, some nodes already reached full convergence. This is because the three processes are concurrent and thus, on typical networks, full convergence time is smaller than in the worst case scenario. In conclusion, note the small group of nodes which reach self convergence in only one time unit. This is a minimal fraction of leaf nodes produced by the Erdős generator.

VII. Tuning Babel with Centrality

Out of the many contexts in which our algorithm can be exploited, we focus on advantages it can give to optimize DV routing protocols in wireless mesh networks. Consider for instance that routing protocols, to perform link-sensing in wireless networks, define a timer t_{H} used by each node to control the generation of link-level HELLO messages. This timer is crucial for a fast re-convergence in case of failures. A trade-off must be found between a short timer, which guarantees fast detection of link failures but subtracts link capacity to data traffic, and a long timer, that is more resource-aware but makes route convergence slow.

The authors of [6] introduce an optimization of t_{H} based on betweenness centrality. Initially they define the average overhead per link (O_H) when every node is configured with the same t_{H}. Then they compute the average estimated loss due to a node failure for the same case (L_H), and they show that keeping O_H constant, the average loss can be reduced if t_{H} is configured per-node as follows:

$$t_{\mathrm{H}}(i) = \frac{\sqrt{d_i}}{\sqrt{b_i}} t_{\mathrm{H}} \frac{\sum_{j=1}^{N} \sqrt{b_j d_j}}{\sum_{j=1}^{N} d_j} \propto \frac{\sqrt{d_i}}{\sqrt{b_i}} \tag{2}$$

Where b_i and d_i are the betweenness and degree of node i. In practice, if a node knows d_i and b_i for all nodes, it can auto-tune its $t_{\mathrm{H}}(i)$ to achieve a convergence time distribution that minimizes the average network disruption after a node failure, keeping a constant signalling overhead in the network.

The authors apply this technique to the OLSR protocol but, in principle, it can be applied to any link-state protocol where every node is aware of the whole topology. Conversely it cannot be applied to DV routing protocols because, in this latter case, nodes have a limited topological knowledge and cannot compute Eq. (2). Algorithm 2 does not mandate nodes to know the entire topology and it is fully distributed, therefore implementation on a DV routing protocol is straightforward and we effectively integrated it with the Babel protocol [15], a well known DV routing protocol[2].

We implemented the distributed centrality computation algorithm in `babeld`, the open source implementation of Babel, in order to verify that centrality can be correctly computed. The evaluation strategy and performance metrics are those proposed by the authors of [6]; here we just briefly review them while a detailed description is provided in the original paper. We run our code in an emulated network using the Mininet platform, at time t_f we trigger the failure of node k, and we let all routing tables stabilize again; we repeat this procedure for a subset of $N_f \leq N$ nodes. Note that generally $N_f < N$ because some nodes are irrelevant (their centrality is zero) or they badly partition the graph so that it is not possible to route around the failed node. During experiments we dump the routing tables $RT_i^j[d] = nh$ every 0.5s: a dump contains the matching between a destination d and the next hop nh for node i at time t_j (only one path is used in Babel). When the emulation is over, we group the dumped routing tables according to timestamps, next we recursively navigate each group to take a snapshot of all shortest paths from every source s to any destination d. Then, we call L_j the number of broken shortest paths that, for $t_j > t_f$, are incomplete or still pass through node k. We call $L_{babel}(k) = \sum_j L_j$ the total loss value when the emulation runs with the original `babeld` and $L_{cent}(k)$ the same value but computed using the optimized

[2]Eq. (2) uses *betweenness* centrality, while our approach computes *load* centrality. However, in a mesh network links are weighted by their quality (with any metric the protocol supports) which makes it hard to have multiple paths with the same exact weights, therefore, in real world mesh networks load centrality converges to betweenness centrality. This said, we do not attempt any comparison with [6], if not for else because comparing a DV and a link-state protocol goes well beyond comparing centrality metrics.

timers. Finally we compare the two approaches computing the relative loss value averaged over all possible failures as:

$$L = 1 - \frac{\sum_{k=0}^{N_f} L_{cent}(k)}{\sum_{k=0}^{N_f} L_{babel}(k)} \quad (3)$$

If $L > 0$, then the tuned version of `babeld`, averaged over N_f failures, produces less losses compared to the non-modified version, always keeping the same overhead due to control messages. We test the protocol on several topologies extracted from real world networks. Two of them are the same ones used and published by authors of [6], and are topologies of two large scale wireless mesh networks. We were able to collect two more real networks topologies analyzing information provided by the FreiFunk German community network (CN). FreiFunk is an umbrella name that gathers together hundreds of wireless CNs in Germany: some of them are made of few nodes, some other are made of hundreds, anyway all of them are mesh networks used to offer Wi-Fi connectivity. Information on these network topologies is freely available (with some effort to understand the format) from the community website[3]. We use two topologies that are heterogeneous networks, with a mix of wireless and wired links inside a single routing domain. Finally, we also use two others extracted from the well known Topology Zoo [33]; these are 4 wired topologies of similar size that we use to extend our analysis.

Before presenting results, it is worth to discuss some modifications to Algorithm 2 that we had to implement in `babeld` and are in general required for any real DV protocol.

a) Nodes Vs Routers: The common approach of the literature focused on centrality is to treat nodes as sources, targets and forwarders of traffic. In real networks, sources and targets are IP addresses and routers have several interfaces with distinct IP addresses. In our case, we were able to aggregate all route-updates coming from the same node based on the "router-id" field defined by Babel to uniquely identify a router. This field is included in all packets generated by a router and is propagated by all others, therefore we can aggregate the centrality contributions pertaining to different interfaces of the same router and do a mapping between IP addresses and graph nodes.

b) Load Estimation: In our implementation, every router generates a unit of traffic $\theta_{s,d} = 1$, but in real networks this value can be arbitrarily tuned. It can be proportional to the dimension of attached subnets (assuming more IPs will generate more traffic) or it can be replaced with an estimation of the real outgoing traffic measured locally. This way load centrality would effectively represent the expected load on the node.

c) Protocol-specific Heuristics: In our tests we used networks that have more than one shortest path with the same weight connecting the same endpoints (s, d). The version of Babel that we used does not support multipath routing: in

[3]See https://api.freifunk.net/, and the visualizer https://www.freifunk-karte.de/.

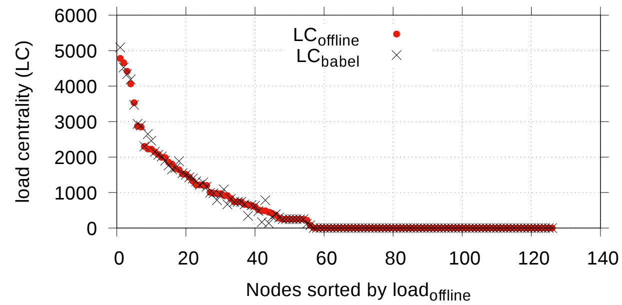


Figure 4. Comparison of LC computed on-line with `babeld` and off-line with networkx on the same topology computed by Babel

these cases Babel performs a tie-break to select one path over another. We also noticed that `babeld` sometimes selects paths that are not minimum weight. This is probably due to an implemented heuristic that prevents to change from one path to another if their weight is similar, just to avoid route flapping. Our algorithm follows choices taken by `babeld`, which is the correct behavior on-line even if the computed LC minimally diverges from the theoretical one. Sec. VII-A further details and explains this issue.

Finally, note that if anybody wants to use Eq. (2), the propagated distance vector should also contain the degree of node i. If we drop this requirement, then we can set $t_{\mathrm{H}}(i) = \frac{\sqrt{t_i}}{\sqrt{b_i}} K$ for some constant K. This will still optimize the timers and keep a constant level of global overhead, but it will not produce exactly the same overhead O_H of the default configuration. In return, it greatly simplifies the protocol as long as every node can take decisions based only on its own centrality. For our purpose we used the correct value of O_H, generated with $t_{\mathrm{H}} = 1s$.

A. Experimental Results

Babel is an event-driven protocol, where messages are sent in reaction to detected changes and expiration of local timers. This, together with the heuristics mentioned in the previous section, introduces slight differences in the centrality computation compared to the ideal protocol studied in Sec. VI. Fig. 4 presents a validation of our implementation in `babeld` and reports a comparison between empirical and theoretical LC values computed for all nodes in a network. On one hand, LC has been computed on-line using the modified `babeld` and, on the other hand, running Python networkx libraries off-line on the same topology built by `babeld` and saved in JSON format. The network we use is ninux, and `babeld` runs in an emulated network with the same topology and characteristics of ninux[4]. First of all we verified that the sum of all LC values is identical in both cases, which means that Babel never uses a minimum weight path that is longer (in terms of hops) than

[4]ninux, as all the other networks we mention, are production networks, thus running experiments like these in the real network is not conceivable even having access to the network itself. Instead, as mentioned already, we use Mininet to emulate the network and run a real instance of the software developed in the virtual nodes/routers in Mininet.

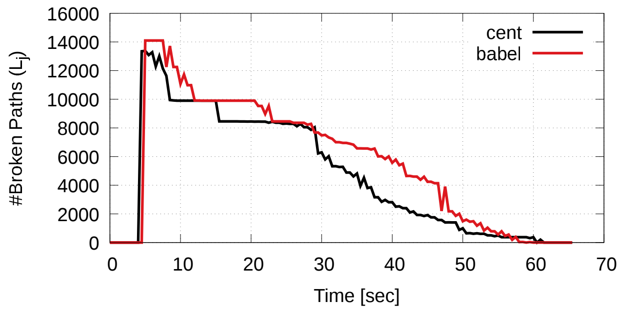


Figure 5. Comparison of the network loss in ninux topology after the failure of one of the most central nodes

the shortest one computed by networkx. Next, we noticed that nodes' rankings are not exactly the same, but very similar.

Fig. 5 reports the number of broken paths vs. time after one of the most central nodes of the ninux topology has failed around time $t = 5$ s. A path is said to be broken if it contains a node with an invalid next-hop. Babel with centrality reacts slightly faster, but above all recovers more routes in less time compared to standard Babel. Thus we achieve a resilience gain without adding (almost) any cost, as long as the complexity of LC computation is minimal: in fact, the signalling overhead in terms of number of messages is constant, while only messages' dimension increases marginally.

To get exhaustive results, we run the modified version of `babeld` in a total of 8 emulated networks representing real topologies. Fig. 6 and Tab. III report the results summary. Fig. 6 compares $L_{babel}(k)$ and $L_{cent}(k)$ where nodes $k = 1, 2, \dots, 15$ are the 15 most central ones for each topology. The chosen networks are ninux (Fig. 6a), Graz (Fig. 6b), and Ion (Fig. 6c), because they well represent different classes of networks; however results would not change significantly selecting other networks. As we can see, in general L_{cent} is smaller than L_{babel}, but sometimes the fine-tuned timers' frequency does not provide any gain, or even leads to a small loss. However, a close look to Tab. III which reports the mean loss reduction for all the 8 considered networks, reveals that averaging over all possible failures we obtain a global gain, ranging from 3% up to 13% depending on the topology; still it is always a clear advantage in favour of tuning timers based on centrality.

In general, wireless and heterogeneous networks achieve larger gains compared to wired and uniform networks due to structural properties of the network graphs. In fact, the optimization level that can be achieved exploiting centrality strongly depends on the array of values of b_i and d_i and on the availability of alternative paths to route around a failure. Consider the extreme case of a ring network, or in general an n-regular network over a torus: in such networks all nodes have same degree and centrality and Eq. (2) returns the same value for all timers. In these cases no optimization is possible.

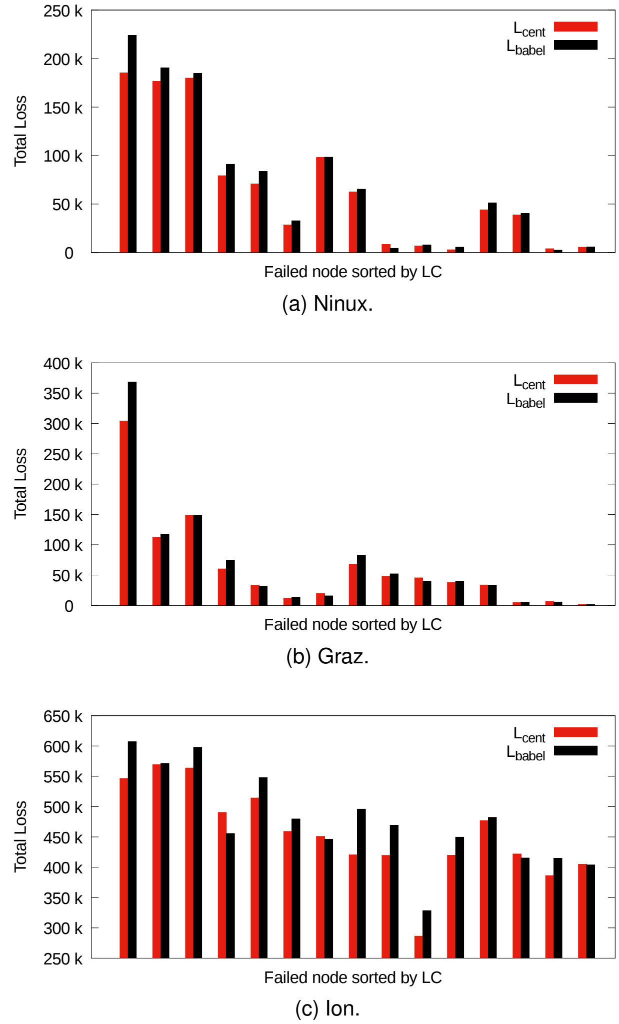


Figure 6. Comparison of the loss induced by the failure of the 15 more central nodes in ninux (a) Graz (b) and Ion (c) when standard Babel is used (L_{babel}) or the modified version is used (L_{cent})

VIII. Conclusions

After decades of research on network protocols, there is no accepted way to tune protocol parameters according to position (and thus importance) of nodes in the network. Centrality is a key instrument to address this issue, make nodes network-aware and differentiate their behavior to achieve better scala-

Table III
Loss reductions in real networks

Network	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	N_f	**Loss Reduction**	**Type**
Interoute	110	148	63	8.37%	Wired
Ion	125	146	58	3.10%	Wired
GtsCe	149	193	98	6.05%	Wired
TataNld	145	186	68	7.34%	Wired
Ninux	126	147	17	10.65%	Wireless
FFGraz	141	200	19	13.11%	Wireless
Auerbach	123	223	70	11.29%	Heterogeneous
Adorf	123	225	65	13.27%	Heterogeneous

bility.

Betweenness Centrality (BC) has been used to approach several problems related to networks but so far, research focused on BC to improve protocols was hampered by the lack of a usable, fully distributed algorithm for BC computation. In fact, among existent algorithms there are those requiring a full topological knowledge, those that are distributed but only approximated and those which are exact and distributed but applicable only on special topologies (like DAGs or trees). We considered *Load Centrality* (LC), a metric coinciding with BC in the most common case (lack of multi-path routing), which however in general better embodies the idea of network load.

This paper presents, to the best of our knowledge, the first algorithm for the exact computation in bounded time of LC in a generic graph. We show that if there is a routing protocol already in place, it can exploit the existent information to compute centrality in a distributed way, otherwise it can be directly integrated with minimal modification into a distance-vector (DV) routing protocol. We demonstrated its convergence, the worst case convergence time, and we confirmed theoretical results with computer simulations. Finally, we provided a direct use-case implementing the distributed algorithm in Babel, a widely used standard DV protocol, showing it can tangibly improve the convergence time in case of nodes' failure for all tested topologies, taken from real networks.

We believe there are many more applications that can benefit from our approach and we can propose improvements as well. Among our future works, we will consider how to integrate the same technique in different routing schemes, such as the so called "source-routed", used by several protocols for mesh networks, or in the widely known "path vector", used by Border Gateway Protocol (BGP).

REFERENCES

[1] D. Katsaros, N. Dimokas, and L. Tassiulas, "Social Network Analysis Concepts in the Design of Wireless Ad Hoc Network Protocols," *IEEE Network*, vol. 24, no. 6, pp. 23–29, Nov. 2010.

[2] M. Kas, S. Appala, C. Wang, K. M. Carley, L. R. Carley, and O. K. Tonguz, "What if Wireless Routers were Social?" *IEEE Wireless Communications*, vol. 19, no. 6, pp. 36–43, Dec. 2012.

[3] U. Brandes, "On Variants of Shortest-Path Betweenness Centrality and their Generic Computation," *Social Networks*, vol. 30, no. 2, pp. 136–145, May 2008.

[4] P. Pantazopoulos, M. Karaliopoulos, and I. Stavrakakis, "Distributed Placement of Autonomic Internet Services," *IEEE Trans. Parallel Distrib. Syst*, vol. 25, no. 7, pp. 1702–1712, Jul. 2014.

[5] S. Dolev, Y. Elovici, and R. Puzis, "Routing Betweenness Centrality," *J. of the ACM (JACM)*, vol. 57, no. 4, pp. 25:1–25:27, Apr. 2010.

[6] L. Maccari and R. Lo Cigno, "Pop-Routing: Centrality-Based Tuning of Control Messages for Faster Route Convergence," in *IEEE Conf. on Computer Communications (INFOCOM)*, Apr. 2016, pp. 1–9.

[7] A. Vázquez-Rodas and L. J. de la Cruz Llopis, "A centrality-based topology control protocol for wireless mesh networks," *Ad Hoc Networks*, vol. 24.B, pp. 34–54, Jan. 2015.

[8] L. Baldesi, L. Maccari, and R. Lo Cigno, "On the Use of Eigenvector Centrality for Cooperative Streaming," *IEEE Communications Letters*, vol. 21, no. 9, pp. 1953–1956, 2017.

[9] P. Zilberman, R. Puzis, and Y. Elovici, "On network footprint of traffic inspection and filtering at global scrubbing centers," *IEEE Trans. on Dependable and Secure Comput.*, vol. PP, pp. 1–16, Oct. 2015.

[10] L. Maccari, Q. Nguyen, and R. Lo Cigno, "On the Computation of Centrality Metrics for Network Security in Mesh Networks," in *IEEE Global Communications Conf. (GLOBECOM)*, Dec. 2016, pp. 1–6.

[11] L. Maccari and R. Lo Cigno, "Betweenness estimation in OLSR-based multi-hop networks for distributed filtering," *Jou. of Computer and System Sciences*, vol. 80, no. 3, pp. 670–685, May 2014.

[12] R. Jacob, D. Koschützki, K. A. Lehmann, L. Peeters, and D. Tenfelde-Podehl, "Algorithms for Centrality Indices," in *LNCS Vol. 3418: Network Analysis*, U. Brandes and T. Erlebach, Eds. Springer, 2005.

[13] U. Brandes and C. Pich, "Centrality Estimation in Large Networks," *Int. J. of Bifurcation and Chaos*, vol. 17, no. 7, pp. 2303–2318, Jul. 2007.

[14] K. I. Goh, B. Kahng, and D. Kim, "Universal Behavior of Load Distribution in Scale-free Networks," *Physical Review Letters*, vol. 87, no. 27, pp. 1–4, Dec. 2001.

[15] J. Chroboczek, "The Babel Routing Protocol," RFC 6126, Apr. 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6126.txt

[16] R. Puzis, M. Tubi, Y. Elovici, C. Glezer, and S. Dolev, "A Decision Support System for Placement of Intrusion Detection and Prevention Devices in Large-Scale Networks," *ACM Trans. Modeling Computer Simulation (TOMACS)*, vol. 22, no. 5, pp. 1–26, Dec. 2011.

[17] U. Brandes, "A Faster Algorithm for Betweenness Centrality," *J. of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[18] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes, "Heuristics for Speeding Up Betweenness Centrality Computation," in *ASE/IEEE Int. Conf. on Social Computing and Int. Conf. on Privacy, Security, Risk and Trust*, Sep. 2012, pp. 302–311.

[19] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating Betweenness Centrality," in *5th Int. Conf. on Algorithms and Models for the Web-Graph (WAW'07)*, Dec. 2007, pp. 124–137.

[20] R. Geisberger, P. Sanders, and D. Schultes, "Better Approximation of Betweenness Centrality," in *Meeting on Algorithm Engineering & Expermiments*, Jan. 2008, pp. 90–100.

[21] Y. Lim, D. S. Menasché, B. Ribeiro, D. Towsley, and P. Basu, "Online Estimating the k Central Nodes of a Network," in *Network Science Workshop (NSW)*. IEEE, Jun. 2011, pp. 118–122.

[22] A. Maiya and T. Y. Berger-Wolf, "Online Sampling of High Centrality Individuals in Social Networks," in *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, Jun. 2010, pp. 91–98.

[23] M. Riondato and E. M. Kornaropoulos, "Fast Approximation of Betweenness Centrality through Sampling," *Data Mining and Knowledge Discovery*, vol. 30, no. 2, pp. 438–475, Mar. 2016.

[24] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres, "Fast Exact Computation of Betweenness Centrality in Social Networks," in *IEEE Int. Conf. on Advances in Social Networks Analysis and Mining (ASONAM)*, Aug. 2012, pp. 450–456.

[25] E. Bergamini and H. Meyerhenke, "Fully-dynamic Approximation of Betweenness Centrality," in *LNCS 9294: Algorithms – ESA 2015*, N. Bansal and I. Finocchi, Eds. Springer, 2015, pp. 155–166.

[26] E. Bergamini, H. Meyerhenke, and C. L. Staudt, "Approximating Betweenness Centrality in Large Evolving Networks," in *17th SIAM Workshop on Algorithm Engineering and Experiments (ALENEX)*, Jan. 2014, pp. 133–146.

[27] N. Kourtellis, G. De Francisci Morales, and F. Bonchi, "Scalable Online Betweenness Centrality in Evolving Graphs," *IEEE Trans. on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2494–2506, Apr. 2015.

[28] Y. Yoshida, "Almost Linear-Time Algorithms for Adaptive Betweenness Centrality using Hypergraph Sketches," in *20th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, Aug. 2014, pp. 1416–1425.

[29] K. You, R. Tempo, and L. Qiu, "Distributed Algorithms for Computation of Centrality Measures in Complex Networks," *IEEE Trans. Autom. Control*, vol. 62, no. 5, pp. 2080–2094, May 2017.

[30] W. Wang and C. Y. Tang, "Distributed Computation of Node and Edge Betweenness on Tree Graphs," in *52nd IEEE Conf. on Decision and Control*, Dec. 2013, pp. 43–48.

[31] ——, "Distributed Computation of Classic and Exponential Closeness on Tree Graphs," in *American Control Conf. (ACC)*, Jun. 2014, pp. 2090–2095.

[32] L. C. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, Mar. 1977.

[33] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Jou. on Selected Areas in Communications (JSAC)*, vol. 29, no. 9, pp. 1765 –1775, Oct. 2011.