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

L. Maccari, L. Ghiro, A. Guerrieri, A. Montresor, R. Lo Cigno

To appear at INFOCOM 2018
Premise

Pop-Routing: Centrality-based Tuning of Control Messages for Faster Route Convergence [*INFOCOM 2016*]

- Computing Centrality requires full graph knowledge
- PopRouting improves resilience and scalability of LS protocols. And DV??

1. Distributed Algorithm for Centrality
2. PopRouting for DV
1. Distributed algorithm for Centrality computation
2. Convergence Study
3. Optimization based on Centrality
Betweeness Centrality  [Brandes 2001]

“Fraction of global shortest paths passing through given vertex”

\[
\overline{BC}(v) = \frac{2}{n(n-1)} \sum_{s,d \in V} \frac{\sigma_{sd}(v)}{\sigma_{sd}}
\]

Where
- \(\sigma_{sd}\) number of minimum weight paths between vertex \(s\) and vertex \(d\)
- \(\sigma_{sd}(v)\) number of those paths crossing \(v\)

Load Centrality (BC)

\[
\overline{LC}(v) = \frac{2}{n(n-1)} \sum_{s,d \in V} \theta_{s,d}(v)
\]
Algorithm working principle

Recursive Aggregation of Partial Centrality Contributions (for node D)

Distributed algorithm for Centrality computation
Theoretical Analysis

• Assumptions
  • DV generation frequency = $\delta s$
  • Propagation time on $x$ hops = $x \times \delta s$; (diameter = $D$)
  • Algorithm starts after Routing Convergence;

• Thesis
  The slowest node computes its load after $\delta(D - 1)s$

• If nodes disseminate their index in the network, then dissemination complexity linear with $2 \times D$
On top of DV algorithm

1. **Init:**
   
   ```
   RT[v].m = RT[v].load = 0;
   RT[v].NH = [];
   RT[v].loadIn = [];
   ```

2. **Repeat every δ s:**
   
   ```
   foreach d ∈ RT() do
     loadOut = 1;
     foreach u ∈ RT[d].loadIn() do
       loadOut += RT[d].loadIn[u];
     send ⟨d, RT[d].m, RT[d].NH, loadOut, RT[d].load⟩ to neighbors;
   ```

3. **on receive:**
   
   ```
   (d, m, NH^u, loadOut^u, load) from u do
     /* Bellman-Ford */
     if d ∉ RT() OR m + C[u] < RT[d].m then
       RT[d].NH = [u];
       RT[d].m = m + C[u];
     else if m + C[u] == RT[d].m then
       RT[d].NH.append(u);
     /* Manage load contributes */
     if u ∈ NH^u then
       RT[d].loadIn[u] = loadOut^u / |NH^u|;
     else
       RT[d].loadIn.remove(u);
     /* Load indexes propagation */
     if u ∈ RT[d].NH then
       RT[d].load = load;
     /* Own load update */
     RT[v].load = 0;
     foreach d ∈ RT() - {v} do
       foreach u ∈ RT[d].loadIn() do
         RT[v] += RT[d].loadIn[u];
   ```

**Routing and algorithm in parallel!**

**Zero Load and no incoming contributions**

**Aggregation of received contribs**

**Splitting**

**Others Load update**

**Own Load Computation**

Distributed algorithm for Centrality computation
1. Distributed algorithm for Centrality computation

2. Convergence Study

3. Optimization based on Centrality
### METRICS

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{NH}$</td>
<td>“Next-hop” set convergence time</td>
</tr>
<tr>
<td>$T_{sl}$</td>
<td>Self load index conv. time</td>
</tr>
<tr>
<td>$T_l$</td>
<td>Time to know all nodes’ load indexes</td>
</tr>
</tbody>
</table>

- Python Simulator
- 2 Synthetic graph models: Erdős and the Barabási-Albert
- Growing diameter in range [3,7]
- For each model: 40 graphs with 1000 nodes each
Less than $2 \times D$ because of parallel processes
1. Distributed algorithm for Centrality computation

2. Convergence Study

3. Optimization based on Centrality
Centrality based Babel

• Mininet Emulator + NePA TesT [WONS 2016]
• Real network topologies
• Comparison of Network Disruption after a node failure* in networks with/without Pop-Optimized Babel

| Network   | $|\mathcal{V}|$ | $|\mathcal{E}|$ | $N_f$ | Type        |
|-----------|-------------|-------------|-------|------------|
| Interoute | 110         | 148         | 63    | Wired      |
| Ion       | 125         | 146         | 58    | Wired      |
| GtsCe     | 149         | 193         | 98    | Wired      |
| TataNld   | 145         | 186         | 68    | Wired      |
| Ninux     | 126         | 147         | 17    | Wireless   |
| FFGraz    | 141         | 200         | 19    | Wireless   |
| Auerbach  | 123         | 223         | 70    | Heterogeneous |
| Adorf     | 123         | 225         | 65    | Heterogeneous |

* $N_f$: number of nodes that, once interrupted, do not partition the network
Network Disruption

Optimization based on Centrality
Optimization based on Centrality
## Loss Reductions in Real Networks

| Network     | $|V|$ | $|E|$ | $N_f$ | 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 |
Conclusions

1. First distributed algorithm to compute Centrality on a generic graph
2. Minimal effort to compute Centrality if Routing is provided
3. Otherwise, it is easy to integrate it with a DV protocol, as shown with Babel
4. Direct application: for instance, we exploited it to improve scalability and resilience of routing protocols

Optimize other timers, integrate with “source-routed” or “path-vector” protocols
Thank you