Yggdrasil routing scheme as a basis for large-scale decentralized mesh networks

Oleksii Pestov[1], Halyna Kyrychek[1] and Mariia Tiahunova[1]

[1] *National University "Zaporizhzhia Polytechnic", 64 Zhukovsky St., Zaporizhzhia, 69063, Ukraine*

Abstract

In this paper, the scalability of the experimental Yggdrasil routing scheme regarding its logic, hop limit, CPU usage, and memory footprint is discussed. Experiments with the routing daemon are conducted on large- and small-scale topologies using meshnet-lab and coreemu-lab environments. The experiments show Yggdrasil to significantly outperform in hop limit other widely used mesh routing protocols, such as OLSR and Babel, give an insight into system resources usage trends, as well as reveal several caveats of Yggdrasil-based networks and a potential way of mitigating them. Taking into account the carried out research, Yggdrasil can be considered a valid and promising basis for the deployment of large-scale decentralized mesh networks of independent nodes as an alternative to traditional centralized hierarchical networking.

Keywords

decentralized mesh networks, mesh routing, Yggdrasil routing scheme, scalability

1. Introduction

Nowadays centralized software-defined networking is becoming increasingly popular with Internet service providers (ISP) due to its remarkable benefits in automated remote hardware configuration, monitoring, control, ease of deployment, use of "white box" hardware [1], etc. That said, the mentioned benefits come with a trade-off at the cost of network reliability and disruption resilience, as centralization inevitably adds singular points of failure to the system. A consequence of this is the relatively recent infamous Kyivstar network collapse that occurred on December 12, 2023, resulting in several days of downtime with a complete lack of customer connectivity. Numerous services throughout Ukraine relied on Kyivstar and could not function until the damage was eventually mitigated.

This incident shows how unreliable centralized hierarchical communication systems may be and calls for a different approach – mesh networking, where the line between routing and terminating (end) devices becomes blurry. The idea of this paradigm lies in every node of the network being able to function independently from any other, establishing connections with other nodes nearby (peering), and collaborating in an effort to route traffic for each other, rather than relying on a central authority for this.

While the concept of mesh networking is not new and a number of routing protocols have been developed (OLSR, Babel, B.A.T.M.A.N. and its derivatives) and successfully used (e.g. the Freifunk initiative and other community networks [2]), there's yet to be one that would scale beyond small local networks and do so efficiently, as in use the least amount of system resources needed for functioning, so that it could be deployed on relatively low-cost low-power devices. Another concern with such networks is the confidentiality of user data as it's forwarded through the network and potentially eavesdropped on by intermediate nodes [3].

ICST-2024: Information Control Systems & Technologies, September 23-25, 2023, Odesa, Ukraine.
apestov02@gmail.com (O. Pestov); kirgal08@gmail.com (H. Kyrychek);
mary.tyagunova@gmail.com (M. Tiahunova)
0009-0002-6092-3301 (O. Pestov); 0000-0002-0405-7122 (H. Kyrychek); 0000-0002-9166-5897 (M. Tiahunova)

2. Theory and related works

Most research in the field of large mesh network routing focuses on its implementation for IoT and wireless sensory networks in particular, optimizing energy consumption and range at the cost of throughput and security. For example, a CottonCandy [4] based network works by nodes arranging themselves into a spanning tree from a designated Internet gateway node as the root, while also avoiding collisions with recursive data requests during duty cycles. While this approach works well for small amounts of periodically collected sensory data, it's not fitting for spontaneous node to node communication in a general-purpose network. Moreover, having a strictly designated root node adds a single point of failure to the network, defeating the purpose of decentralization. Yamamoto et al. [5] propose the VORTEX routing protocol with an approach based on opportunistic routing on a hierarchical tier structure established during the initialization phase. The network is treated as a "black box" by communicating node pairs, and intermediary nodes simply forward the packets along the hierarchy until they reach the recipient taking multiple paths for better reliability. Thus, the need to execute route discovery first is omitted, however the network ends up being flooded with extra traffic, which would severely diminish its performance on a large scale. Prabu et al. [6] present a novel approach to mesh routing by using the concept of "heat trails" and representing it using bloom filters [7] in several intensity gradient levels. Nodes randomly probe the network until they eventually reach destinations node's "heat", then follow it in the direction of increasing intensity. When the destination is reached, a "heat trail" is formed along the probed path, so that other nodes looking for the previous two may stumble upon it and follow to their destination, partially reusing the route. While in use, the discovered route is gradually optimized. The main problem with this approach is, once again, its large-scale performance due to random probing – with a low amount of "heat" gradient levels it could potentially be needed to probe the entire network to find the destination, and increasing it would lead to increased memory usage. A potential solution to the aforementioned problems could be in the use of the experimental Yggdrasil routing scheme [8], the main point of which (the current latest version being v0.5.5) is in the use of Ed25519 public key [9] based addressing and a self-arranging spanning tree of the network. The latter is similar to CottonCandy, except the root of the tree is the node with the lowest key value instead of a designated gateway, and thus can be automatically replaced in case of a failure. Bloom filters are present on every on-tree link and represent a set of node keys reachable through said link. Node lookups are done on-demand using broadcasts, that eventually get culled with these bloom filters (*O(n)* computational complexity, where *n* is the number of on-tree links of a given node). The result of such a lookup is the tree coordinates of the receiving node relative to the current root of the network, by which the traffic gets sent. On every subsequent node opportunistic greedy routing [10] is applied to figure out the direction the traffic takes; in other words, the packets are sent over whatever single link brings them closer to specified tree coordinates (*O(n)* computational complexity, where *n* is the number of peers of a given node). This part of the process is similar to VORTEX in that route discovery is omitted, instead forwarding the data with local decisions based on a hierarchical structure, but not necessarily following it.

The described approach has a number of additional advantages: nodes generate addresses for themselves independently; transparent asymmetric end-to-end encryption; source/destination verifiability with cryptographic keys used for addressing. Network congestion is managed automatically at each node using a form of fair packet queuing, which attempts to balance traffic flows between nodes equally where possible. The main disadvantage of Yggdrasil's approach is that traffic does not always take the shortest possible path, introducing higher latency, which can, however, be accounted for. Link quality is also not considered outside of priorities between multiple peerings to the same. Every node of the network only holds in memory a 1KB bloom filter for every on-tree link as well as 32B keys of its direct peers and their ancestors on the path to the current root. This means a particular node's memory usage should only depend on the number of links and the distance from the current root. As for CPU usage, notable spikes are expected on the transmitting and receiving nodes due to encryption and decryption taking place respectively, as well as some positive correlation with the number of links. Due to the Yggdrasil routing scheme being relatively new and still in the alpha stages of development, there aren't many works that cover or even mention it. One paper [11] presents benchmark results for using Yggdrasil's overlay testbed network as a NAT traversal tool for the purposes of remote control, but no scalability

testing of the routing scheme itself. Another work [12] compares Yggdrasil with a variety of other mesh network routing protocols in a number of benchmarks, only one of which considers scaling beyond 100 nodes. It is also worth noting that in the aforementioned benchmark node reachability is measured only once with a limited packet arrival deadline. This approach is not exactly fitting for Yggdrasil and on-demand routing in general, as the first packet always has a significantly longer round trip time than the subsequent ones due to node lookup taking place first. The research object of this paper is the evaluation process of the Yggdrasil routing scheme. The research subject is models and methods for evaluating Yggdrasil's scalability regarding the logic, hop limit, CPU usage and memory footprint of the routing daemon. The purpose of this research is to evaluate Yggdrasil's viability as a basis for large-scale decentralized communication networks.

3. Proposed methodology

3.1. Large-scale benchmarking

The purpose of large-scale benchmarking is to determine the performance of Yggdrasil on large networks (50 nodes and above) while recording the system resource (CPU and memory) usage of each node's routing daemon instance. Such testing is meant to give an idea of technical specifications of hardware needed for an Yggdrasil network, as well as to show its hop limit if there is one. Scaling of system resource usage and convergence time with network size is also important when deciding on the viability of the routing scheme.

The tests were conducted using the meshnet-lab environment due to its flexibility, the possibility of emulating large-scale topologies with Linux network namespaces [13], and automating the process with Python. It should be noted, that the host system the tests were conducted on can't handle more than 750 nodes at once, as beyond that point the system runs out of memory and starts to use the swap partition. The benchmarking procedure is as follows:

Prepare a topology with pre-generated JSON files included in meshnet-lab, and limit link bandwidth to 100 Mbit/s. Launch routing daemons on nodes. Launch the system resource monitoring script (records CPU and resident memory usage of routing daemons every second). Wait 30 seconds for nodes to discover each other.

Over the next 30 seconds send out a total of N unique random pings with a minimum hop count of 2 and a deadline of 30 seconds, where N is the number of links in the topology. Wait 15 seconds. Repeat steps 5-6 five more times.

Launch the node information collection script (requests Yggdrasil-specific information, such as public keys and tree views, from the routing daemons; needed to determine the root node in result processing). Stop the routing daemons and resource monitoring scripts, and clear the network. Repeat the procedure on a bigger topology until less than 40% of the pings arrive or more than 750 nodes are to be simulated.

The described procedure was conducted on random tree and line topologies (Figure 1) as a more realistic and worst-case scenario respectively. On larger topologies, it was needed to increase the number of iterations in step 7 for the network to reach convergence. Additionally, for the purpose of looking into the memory usage trend on a converged network, prolonged experiments were conducted on the largest viable topology.

A topology is considered viable if it ends up reaching convergence indicated by 100% ping arrival. Another experiment on a random tree topology of 50 nodes with no bandwidth limitation showed the difference in CPU usage for transmitting, receiving and intermediate nodes, as well as nodes not taking part in the transmission.

Traffic was generated for 10 seconds between two distant nodes (001e and 0032, see Figure 1) using the iperf3 utility. Lastly, an experiment on a 750 nodes line topology with imperfect slow links showed Yggdrasil's resilience to such conditions. Link parameters applied here were 10 Mbit/s bandwidth with a 10±5 ms latency. Traffic was generated between the first and last node using the standard ping utility.

The last two tests were performed manually. The benchmarking procedure was automated with a modified version of the "benchmark1" Python script included in meshnet-lab. Resulting data was analysed and plotted using pandas, seaborn and Matplotlib Python libraries [14, 15].

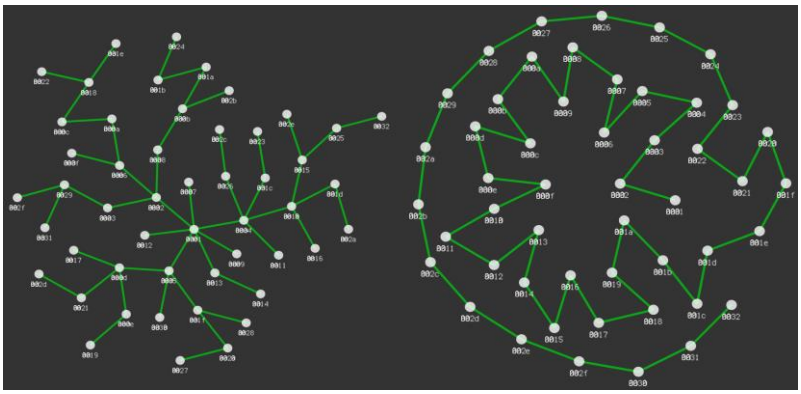

Figure 1: 50 nodes random tree and line topologies

3.2. Small-scale tests

The purpose of small-scale testing is to get an insight into Yggdrasil's performance in more complicated scenarios than a large static network. Due to this, the testing process was more exploratory in nature. Overall, we needed to see how exactly the Yggdrasil spanning tree is arranged, what paths the traffic takes, and how tolerable node mobility is. This information is important in that it exposes potential caveats and flaws of the routing scheme. For this, a tree topology with a known root would be built.

After launching Yggdrasil routing daemons on it, additional links were introduced. The resulting network would then be analysed by querying the routing daemons about their views of the tree and launching various one-sided traffic flows to see the paths they end up taking. Node mobility tests were done by having one roaming node jump between static peers while pinging another node.

Of particular interest here were the cases of a node choosing a new parent upon losing the current one and the network having a roaming root node.

The tests were conducted in the coreemu-lab [16] environment due to it being a "batteries-included" Docker image distribution of the Common Open Research Emulator (CORE) extended with automated testing and node monitoring functions [17]. This software provides a graphical user interface for easy network configuration and interaction with the nodes, as well as multiple node and connection types including a wired and wireless LAN. The latter is particularly useful due to the ability to dynamically form and break connections by simply dragging nodes around during the emulation.

Node movement can also be automated with mobility scripts. Another useful feature of CORE is the visualization of throughput on wired links and wireless interfaces, allowing for traffic flow tracing. The provided Docker image was modified to include the Yggdrasil routing daemon. The following tests are run with pre-generated keys for every node for better control over the Yggdrasil spanning tree.

4. Results

4.1. Large-scale benchmarking

The results of memory benchmarking on the random tree topology are presented in Figures 2 and 3.

Figure 2 confirms the assumption about the memory usage of a node being positively influenced by the number of its links. Figure 3 shows that, despite the increase in overall network size, memory usage for the absolute majority of nodes stays roughly within the same range. Packet arrival for random tree topology stays at an all-time 100%.

Figures 4 and 5 depict the results of the same benchmark conducted on a line topology. Because on line topology every node except the first and the last one has only two links, the hue and size of the points here represent the nodes' positions in the line.

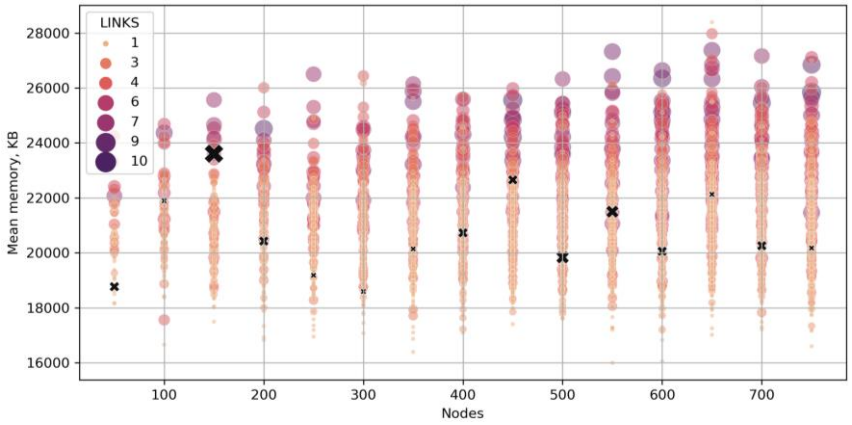

Figure 2: Mean memory usage per node on different network sizes, random tree topology, size and hue represent the number of links a node has, crosses mark root nodes

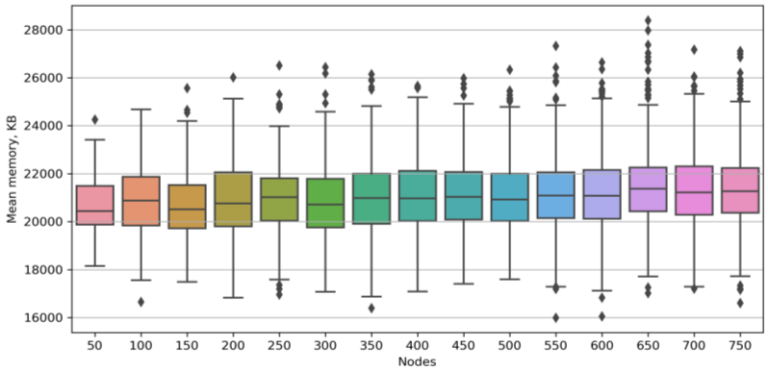

Figure 3: Box plot distribution of mean memory usage per node on different network sizes, random tree topology

Comparing Figures 3 and 5 confirms the assumption about memory usage being positively influenced by network length. This is also confirmed by peak memory usage per topology presented in Table 1.

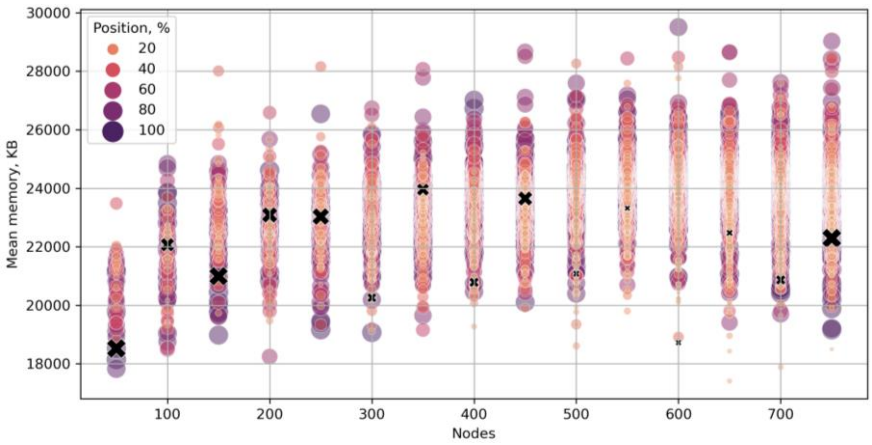

Figure 4: Mean memory usage per node on different network sizes, line topology, size and hue represent the node's position in the line in percent, crosses mark root nodes

This trend is much more noticeable on linear topologies, although for network sizes beyond 300, it stops being as consistent. It is also clear that on either topology being a root node doesn't necessarily result in higher memory usage. In fact, due to nodes keeping in memory only their and their peers' ancestors' keys, higher memory usage can be seen on the nodes further away from the root. That said, outliers are also present, like nodes far from the root with lower memory usage and

the other way around, as well as nodes with a lower link count, but higher memory usage. This can be partially attributed to the randomness of pings. Overall, even on a 750 node line the memory footprint of the routing daemon did not exceed 33 MB, and, following the observed trend, it is expected to rise fairly slowly beyond that.

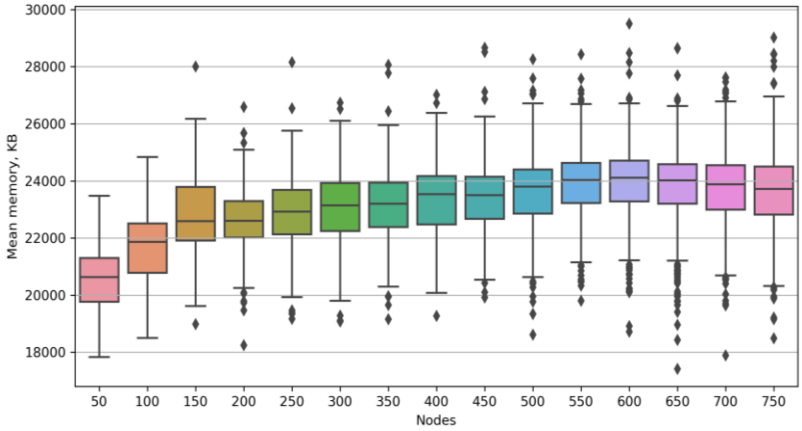

Figure 5: Box plot distribution of mean memory usage per node on different network sizes, line topology

Table 1
Peak memory usage on random tree and line topologies

Net-work size	Peak memory usage, KB		Net-work size	Peak memory usage, KB		Net-work size	Peak memory usage, KB	
	Random tree	Line		Random tree	Line		Random tree	Line
50	28640	26640	300	29300	30300	550	28664	30740
100	27980	27740	350	28812	31840	600	28624	32712
150	29036	29496	400	29004	30512	650	30092	32116
200	27432	28104	450	27736	30812	700	29620	30912
250	28660	29912	500	29428	31168	750	29564	30656

Another dynamic for a linear topology can be observed on a "mean CPU usage vs node position" plot shown in Figure 6. When nodes are arranged in a line, CPU usage scales with its length and is greater the closer to the middle of the line a particular node is. Furthermore, the root node seems to always divide the line into two parts, for which length and middle points are considered separately. This is especially clear on 700 and 450 node lines, where the root node happened to be closer to the middle. On the former, there are significantly fewer nodes to the left of the root than to the right of it, resulting in a massive CPU usage difference while still keeping the trend of middle nodes having a greater CPU usage for both sides. The 450 node topology has a similar case, except this time it's divided into roughly equal parts, allowing it to have a lesser overall CPU usage than the smaller 400 node network. This observation leads to the conclusion that it is more beneficial to keep the root of the network closer to its actual center, as in keeping the number of nodes on every branch roughly similar. One of the ways this can be achieved is purposeful "mining" for lower keys.

Note: CPU usage is recorded with the standard Linux ps utility, in which, according to its manual page [18], this metric is currently expressed as the percentage of time spent running during

the entire lifetime of a process. This is not ideal, but is fitting for this use case, as it does allow the comparison of routing daemon processes.

Figure 7 features the packet arrival ratio over time on a line topology for different network sizes. Line points are slightly shifted on the time axis due to pings having a longer round-trip time on longer topologies. The 700 and 750 node networks never quite reach 100% packet arrival. Upon further investigation, this happens due to the ping utility reporting a "mixed" result to the meshnet-lab environment – on long topologies it needs to wait significantly longer for the initial packet to arrive because of node lookup taking place. This leads to it sometimes sending a second one, which arrives successfully and causes the result to be "50% arrival". Having this information, we can consider packet arrival above 95% to signify a converged network and find out the approximate convergence time presented in Table 2. It should be noted that the measurements were recorded at least 60 seconds apart from each other, so these numbers should not be taken at face value. They can, however, be used to build a convergence time plot shown in Figure 8. The resulting plot suggests a linear relation between network length and convergence time.

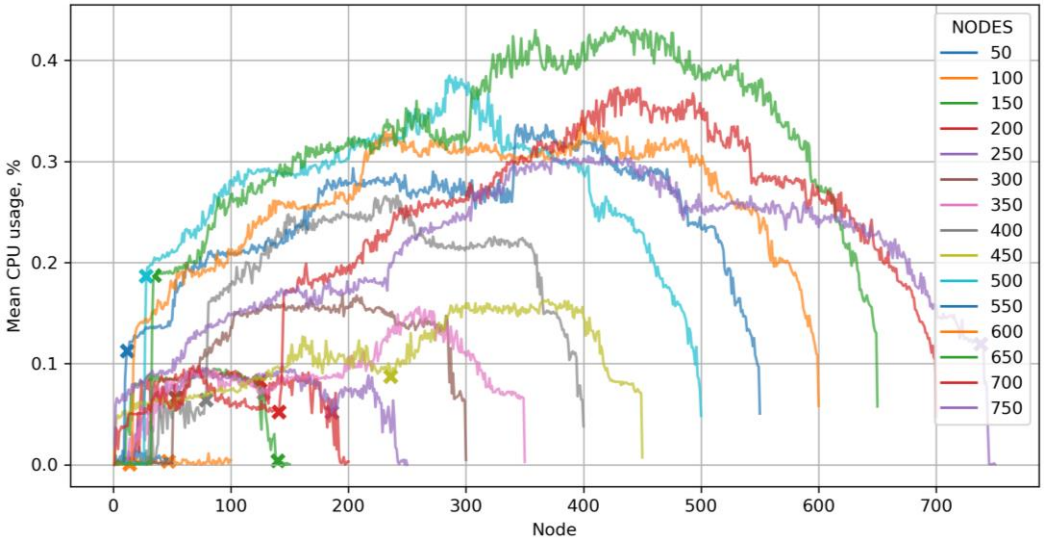

Figure 6: Mean CPU usage for every node in a line topology on different network sizes, crosses mark root nodes

For comparison, the same experiment on a line topology was conducted with different routing protocols (OLSR, Babel) and showed their inability to handle long networks – packet arrival goes below 80% at 150 nodes and below 40% at 450 (see Figure 9).

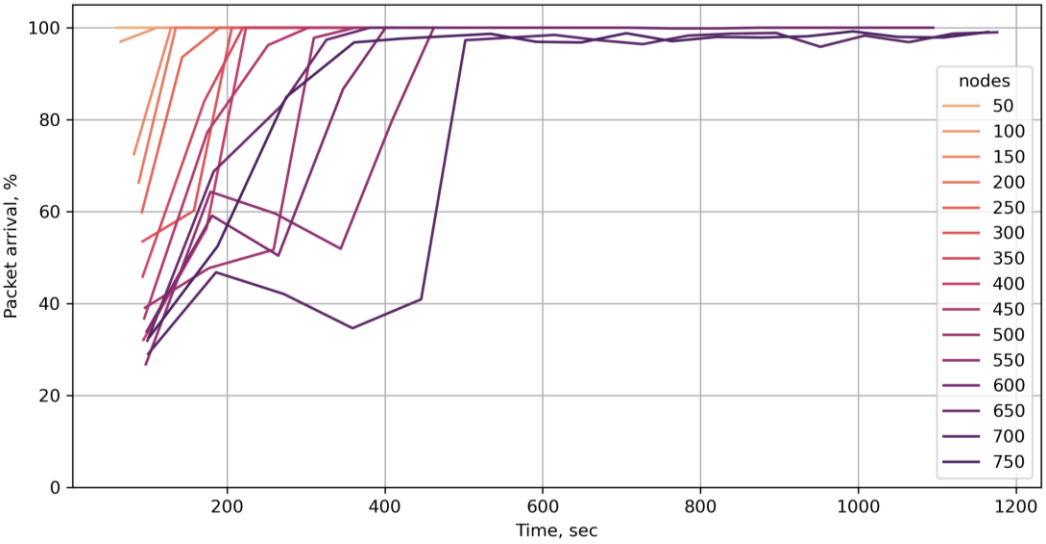

Figure 7: Packet arrival over time for different network sizes, line topology

Extended experiments were also conducted and showed packet arrival to never increase for these protocols.

Memory usage over time on a 750 node line topology is presented in Figure 10. The lines have decreased opacity to indicate a general trend, which suggests constant memory usage after the network reaches convergence. The results of CPU usage measurement on a 50 node random tree topology without bandwidth limitations are featured in Figure 11.

Table 2
Convergence times on a line topology

Network length	Convergence time, seconds	Network length	Convergence time, seconds	Network length	Convergence time, seconds
50	60	300	206	550	462
100	65	350	220	600	401
150	129	400	224	650	326
200	135	450	252	700	502
250	190	500	310	750	361

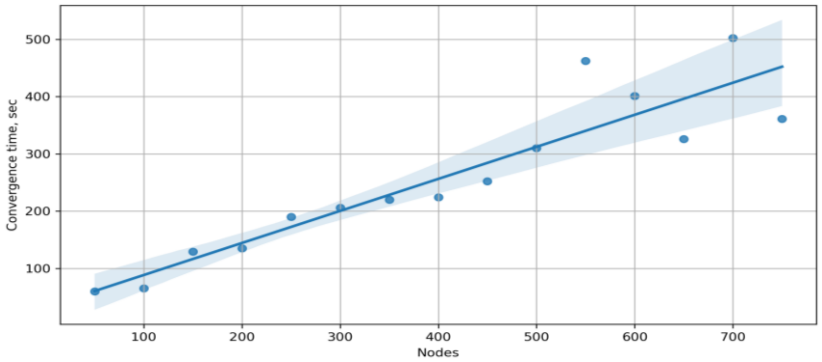

Figure 8: Convergence time on a line topology of different lengths

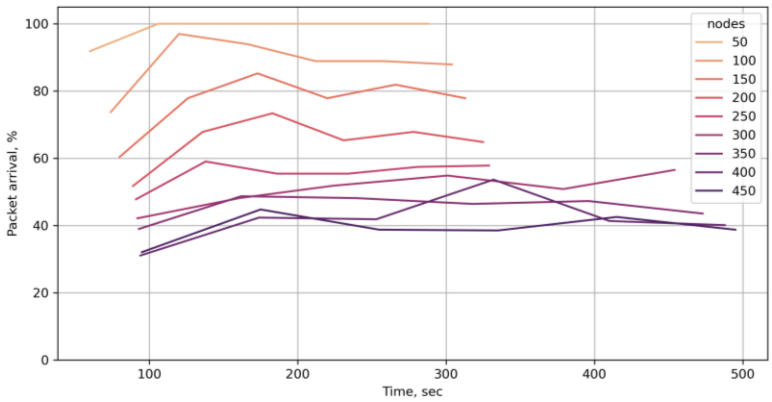

Figure 9: Packet arrival over time for different network sizes, line topology, OLSR protocol

The plot suggests receiving and transmitting nodes to have the greatest and second greatest CPU usage respectively, with intermediate nodes following them and the rest of the nodes not taking part in the transmission, which confirms the assumption mentioned in section 3.1. The CPU usage measurements rise while the transmission is ongoing, and then gradually fall off due to the specifics of the ps utility implementation mentioned earlier.

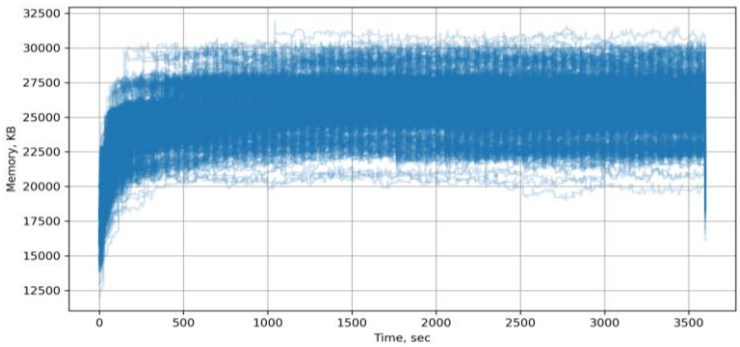

Figure 10: Memory usage over time, 750 node line topology

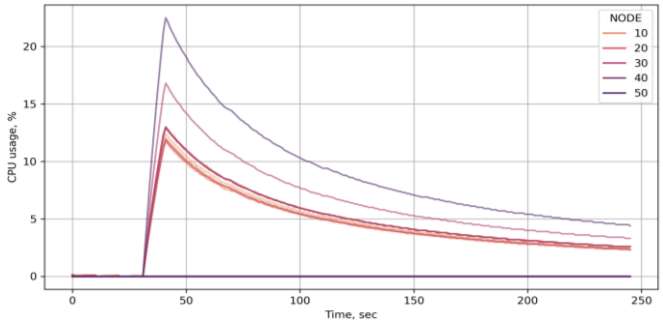

Figure 11: CPU usage over time, 50 node random tree topology, traffic from node 001e to node 0032 (30 and 50 decimal)

The experiment on a 750 node line topology with slow links showed 749 hop communication to be possible still, albeit with a considerable latency of 18 to 20 seconds, which calls for delay tolerant applications to be used with such a network.

The results of large-scale tests demonstrated the lack of observable hop limit with linear scaling of convergence time to network length. Memory usage also scales with network length instead if its overall size, staying within acceptable margins. CPU usage was shown to be influenced by root node placement, opening up a way of optimizing it. Thus, Yggdrasil is confirmed to be viable for use in large mesh networks.

4.2. Small-scale tests

The first small-scale experiment was conducted on a 9 node line topology with node n1 establishing links with several nodes in said line. The network and the corresponding sequence of actions are shown in Figure 12. Purple arrows indicate the structure of the Yggdrasil spanning tree in the form of "child of" relationships, with n9 being the root node. First, n1 is only peered with n5 being its child. Then, peerings with n2 and n7 are established in this exact order. When n1 – n5 link gets severed, n1 consistently chooses n2 as its parent as the link with the longest uptime, despite n7 being closer to the root. While this logic might seem inefficient, it keeps the network more stable by eventually leaving only the most stable links in the spanning tree and thus making it a "backbone" of the network. It also becomes clear how exactly the spanning tree is built – the keys only matter in the choice of the root node, which becomes the point of reference for the rest of the network. The spanning tree is arranged around the root based on the links that are brought up first and taken down last.

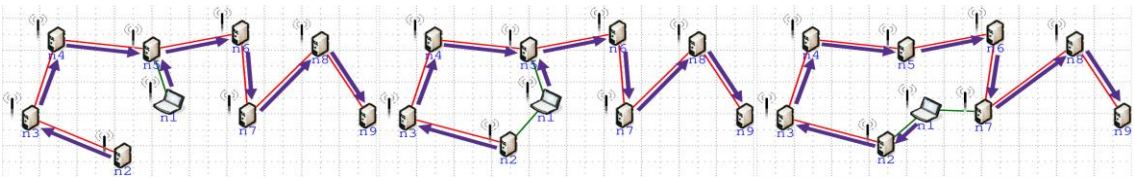

Figure 12: 9 node linear network, node n1 establishes different peerings

The next experiment was conducted on a 13 node network featured in Figure 13. Here, wired links are used for the majority of nodes with unidirectional traffic flows between nodes n1 and n2. Due to greedy routing, these traffic flows end up taking different paths. Off-tree links n12 – n14 and n13 – n15 allow the traffic to skip two nodes and are thus taken. While this feature of traffic flows between two nodes taking different paths may not have been intentional, it is beneficial in providing more bandwidth and mitigating congestion. However, greedy routing has its disadvantages, mainly in only considering single link paths. An example of this is can be seen at the bottom of Figure 13. The network is the same as in the previous example, except nodes n14 and n15 are swapped places. As a result, links n12 – n14 and n13 – n15 no longer provide any immediate benefit from n12's and n13's point of view, thus not being taken. This results into both traffic flows taking the same longer path through the root node. Two experiments regarding node mobility were conducted, one with a leaf node being the roaming one and one with a roaming root on a bigger network. The former involved node n1 constantly jumps between leaf nodes in the network presented in Figure 14.

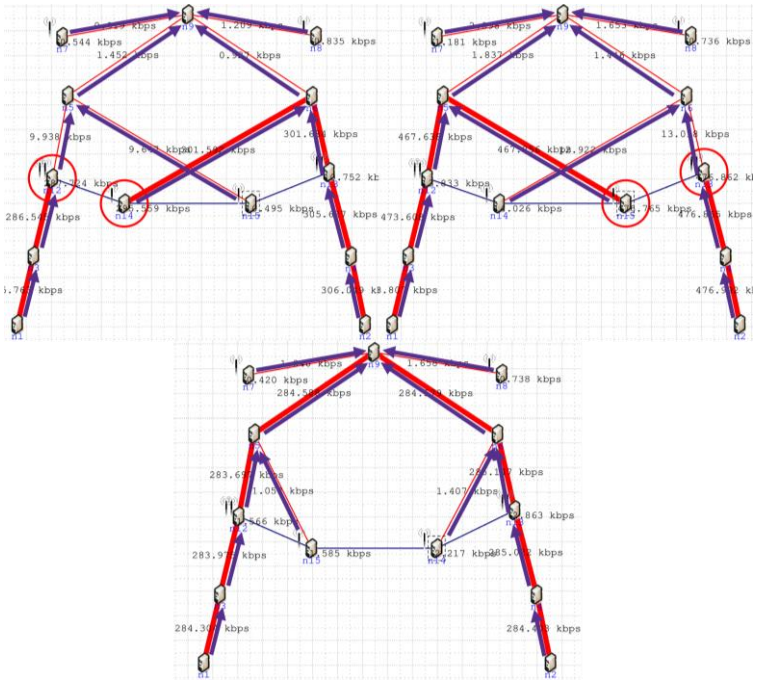

Figure 13: Top: 13 node network, unidirectional traffic flows n1 → n2 and n2 → n1; Bottom: altered

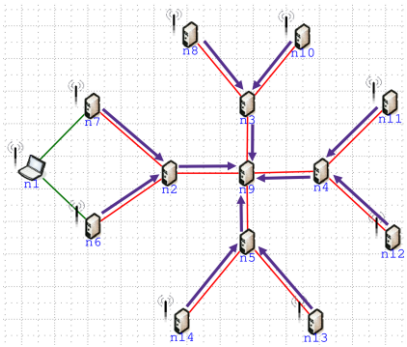

Figure 14: 14 node network, roaming node n1

Such behaviour did not impact the connectivity within the network besides node n1 itself. Indeed, constantly changing peers means constantly changing tree coordinates, thus preventing any prolonged traffic flows to the roaming node or disrupting them with node lookups. Traffic originating in node n1 has a better time reaching its destination given that the destination is a static node, as its tree coordinates do not change after the lookup and throughout the transmission.

To successfully receive traffic, a roaming node has to maintain at least one constant peer for the duration of the transmission, otherwise it would be interrupted with a node lookup. Thus, Yggdrasil tolerates node mobility but does not encourage it and requires nodes to maintain their coordinates for bidirectional data exchange.

A roaming root node, however, is an entirely different case. Figure 15 features a network similar to the previous example, except this time the root node n9 is constantly jumping between pairs of leaf nodes with a period of 10 seconds. This causes the nodes it comes into contact with to change their tree coordinates and propagate this change throughout the network as they should, however, by that time the root comes into contact with different peers, causing a conflicting reordering to propagate as well. This results into nodes having different views on the spanning tree, which, in turn, is the reason of various traffic anomalies seen in Figure 15. On such a small scale the traffic between n18 and n26 takes the correct route for the majority of time, only sometimes being disrupted with anomalies, but it's only reasonable to expect major disruptions in a larger network, especially when new root node candidates pop up in completely different parts of the network. This can be considered a potential attack vector. A possible way to mitigate it is the previously mentioned low key mining, allowing for a constant root to be set up with a pre-generated considerably low key, to the point of mining for a lower one becoming unreasonable and detrimental. It would also make sense to set up several potential roots like this for the sake of redundancy.

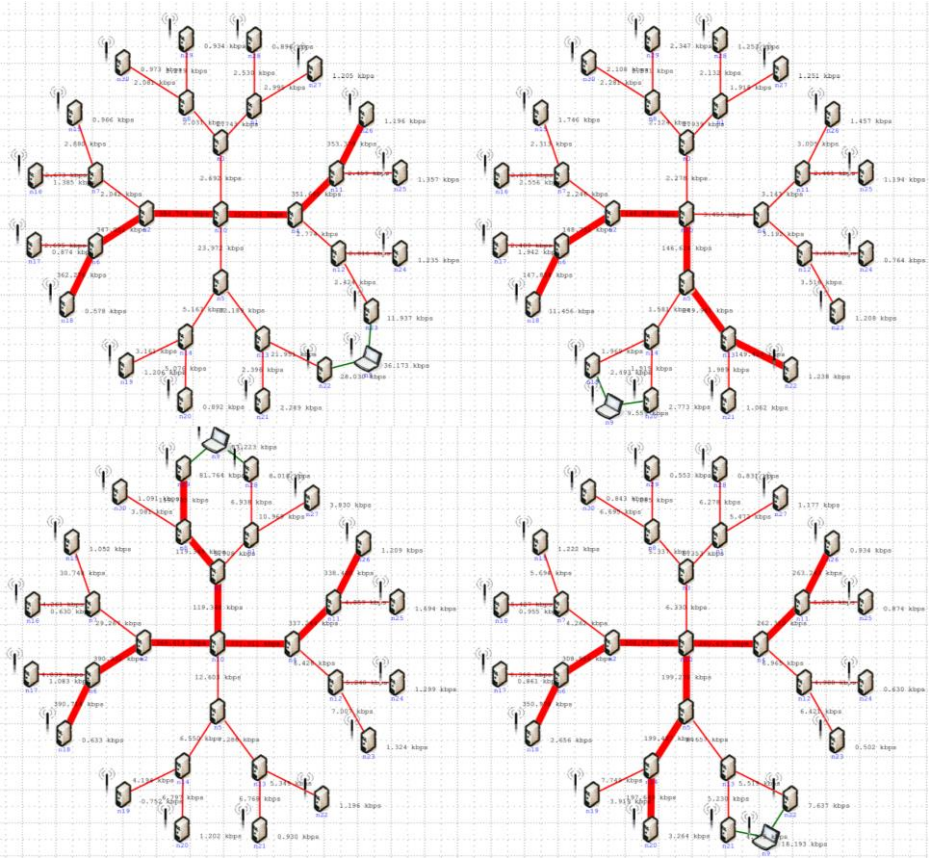

Figure 15: 30 node network, roaming root node n9, normal traffic from n18 to n26 (top left) and its anomalies

Small-scale testing exposed the logic behind spanning tree arrangement and traffic routing, which strives for using the most stable links for the spanning tree and taking shortcuts where possible with the lowest effort from intermediary nodes. Mobility tests showed Yggdrasil to require at least one stable peering for the receiving node during transmission, while otherwise roaming leaf nodes only affect their own availability with eventual reestablishment of it after mobility events cease. Root node mobility was found to be a potential attack vector with deterministic root placement available as a means of mitigation. Thus, Yggdrasil is shown to be fitting for semi-stable networks with no requirement of uninterruptible transmission during mobility events.

5. Conclusion

In this paper the scalability of the experimental Yggdrasil routing scheme in regards to its logic, hop limit, CPU usage and memory footprint of the routing daemon was discussed. The computational complexity of node lookup/routing was evaluated as linear to the number of on-tree links/peers of a given node. A number of experiments were conducted on both large- and small-scale topologies using meshnet-lab and coreemu-lab environments. The results of the experiments showed Yggdrasil to significantly outperform in hop limit other widely used mesh routing protocols, such as OLSR and Babel. The experiments also showed the typical convergence time, as well as memory and CPU usage trends of Yggdrasil and confirmed their scaling with the depth of the network rather than the overall size of it. The memory footprint of the routing daemon never exceeded 33 MB. This makes Yggdrasil a valid and promising basis for the deployment of large-scale decentralized mesh networks of independent nodes as an alternative to traditional centralized hierarchical networking used in ISP networks.

Testing also discovered several caveats in Yggdrasil-based networks, such as increased CPU usage on longer spanning tree branches and traffic anomalies introduced by a roaming root node. Preemptive low key mining for deterministic root node placement was suggested as a means of mitigation for these problems.

Before planning and constructing a real prototype network based on Yggdrasil, a number of other topics have to be researched, like preferred underlying link and physical layer technologies, required hardware, deployment on common hardware, user adoption, etc. Another question of rather significant importance is security, especially with Yggdrasil being an open network – as any node is reachable by any other (in contrast to Carrier-grade NAT employed by ISPs [19]) and there is no single entity in control of the network, malicious activity cannot be mitigated by simply disconnecting bad actors from the network, leaving every node to fend for itself. As for underlying link layer technologies, a possible candidate would be the 802.11s standard [20] with disabled traffic forwarding and deployment of wired links where possible. Further work will be devoted to delving into the aforementioned questions.

References

[1] H. Nishizawa, Architecting Cloud-native Optical Network with Whitebox Equipment, in: Optical Fiber Communication Conference, OSA, Washington, D.C., 2020. doi:10.1364/ofc.2020.w3c.5.

[2] J. A. Greig, Wireless Mesh Networks as Community Hubs: Analysis of Small-Scale Wireless Mesh Networks and Community-Centered Technology Training, J. Inf. Policy 8.1 (2018) 232–266. doi:10.5325/jinfopoli.8.1.0232.

[3] O. R. Rudkovskyi, G. G. Kirichek. Interaction support system of network aplications. In CEUR Workshop Proceedings 2832 (2020) 11-23 URL: https://ceur-ws.org/Vol-2832/paper01.pdf.

[4] D. Wu, J. Liebeherr, A Low-Cost Low-Power LoRa Mesh Network for Large-Scale Environmental Sensing, IEEE Internet Things J. (2023) 1. doi:10.1109/jiot.2023.3270237.

[5] R. Yamamoto, T. Yamazaki, S. Ohzahata, VORTEX: Network-Driven Opportunistic Routing for Ad Hoc Networks, Sensors 23.6 (2023) 2893. doi:10.3390/s23062893.

[6] S. Prabu, M. Maheswari, B. Jothi, J. Banupriya, B. Garikapati, Efficient Bloom Filter-Based Routing Protocol for Scalable Mobile Networks, in: RAiSE-2023, MDPI, Basel Switzerland, 2023. doi:10.3390/engproc2023059075.

[7] F. Grandi, On the analysis of Bloom filters, Inf. Process. Lett. 129 (2018) 35–39. doi:10.1016/j.ipl.2017.09.004.

[8] Yggdrasil Network. 2021. URL: https://yggdrasil-network.github.io/.

[9] J. Brendel, C. Cremers, D. Jackson, M. Zhao, The Provable Security of Ed25519: Theory and Practice, in: 2021 IEEE Symposium on Security and Privacy (SP), IEEE, 2021. doi:10.1109/sp40001.2021.00042.

[10] M. Khaledi, A. Rovira-Sugranes, F. Afghah, A. Razi, On Greedy Routing in Dynamic UAV Networks, y: 2018 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops), IEEE, 2018. doi:10.1109/seconw.2018.8396354.

[11] P.-N. Messan, S. Krupinski, G. Vallicrosa, P. Ridao, F. Maurelli, Evaluation of computer networking methods for interaction with remote robotic systems, 2021 IEEE AFRICON (2021) 1–6. doi:10.48550/arXiv.2110.06385.
[12] B. Reich, Wifi-Ad-hoc Mesh Networks for mobile Systems, Bachelor thesis, Hochschule für Angewandte Wissenschaften Hamburg, Hamburg, Germany, 2024. URL: http://hdl.handle.net/20.500.12738/14753.
[13] D. Schubert, B. Jaeger, M. Helm, Network Emulation using Linux Network Namespaces, Network 57 (2019). URL: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2019-10-1/NET-2019-10-1_11.pdf.
[14] J. P. Mueller, L. Massaron, Python for Data Science for Dummies, Wiley & Sons, Incorporated, John, 2019.
[15] Chapter 5: Matplotlib and Seaborn, in: Data Literacy with Python, De Gruyter, 2023, pp. 117–164. doi:10.1515/9781501518652-006.
[16] L. Baumgartner, T. Meuser, B. Bloessl, coreemu-lab: An Automated Network Emulation and Evaluation Environment, y: 2021 IEEE Global Humanitarian Technology Conference (GHTC), IEEE, 2021. doi:10.1109/ghtc53159.2021.9612475.
[17] J. Ahrenholz, C. Danilov, T. R. Henderson, J. H. Kim, CORE: A real-time network emulator, y: MILCOM 2008 - 2008 IEEE Military Communications Conference (MILCOM), IEEE, 2008. doi:10.1109/milcom.2008.4753614.23
[18] ps(1) - Linux manual page. 2020. URL: https://man7.org/linux/man-pages/man1/ps.1.html.
[19] I. Livadariu, K. Benson, A. Elmokashfi, A. Dhamdhere, A. Dainotti, Inferring Carrier-Grade NAT Deployment in the Wild, y: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, IEEE, 2018. doi:10.1109/infocom.2018.8486223.
[20] A. Flammini, E. Sisinni, F. Tramarin, IEEE 802.11s performance assessment: From simulations to real-world experiments, y: 2017 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), IEEE, 2017. doi:10.1109/i2mtc.2017.7969752.