Submitted to IEEE Access for review on August 20, 2020, current version 1.0.0, date of current version August 20, 2020.

Digital Object Identifier xxxxx

The Streamr Network: Performance and Scalability

PETRI SAVOLAINEN[1], SANTERI JUSLENIUS[2], ERIC ANDREWS[2], MIROSLAV POKROVSKII[2], SASU TARKOMA[1] (SENIOR MEMBER, IEEE) AND HENRI PIHKALA.[2]

[1]University of Helsinki, Department of Computer Science, P.O. Box 68, 00014 University of Helsinki Finland (firstname.lastname@helsinki.fi)
[2]Streamr Network AG, BahnhofStrasse 21, 6300 Zug Switzerland (e-mail: firstname.lastname@streamr.network)

Corresponding author: Petri Savolainen (e-mail: petri.savolainen@ helsinki.fi).

ABSTRACT Decentralized publish-subscribe has become a popular communication pattern in Internet of Things (IoT) and blockchain networks. However, many decentralized publish-subscribe proposals suffer from long and unpredictable message propagation delays and vulnerability to various attacks. In this paper we present and evaluate the Streamr Network, a decentralized publish-subscribe system already in production, that solves the aforementioned problems by enforcing the intended network topology with the help of trackers in order to keep the latencies low and predictable. Our experiments, conducted with up to 2048 nodes placed in Amazon datacenters around the world, show that the message propagation delays in the Streamr network scale logarithmically as a function of the number of nodes, and that they can be estimated using Dijkstra's algorithm with a mean absolute percentage error (MAPE) of 3.5%.

INDEX TERMS distributed, decentralized, scalable, publish-subscribe, publish, subscribe, Internet of Things, IoT, peer-to-peer, P2P, smart contract

I. INTRODUCTION

The publish-subscribe model has emerged as a popular communication pattern in Internet of Things (IoT) networks. The decoupling of data producers and consumers offered by the publish-subscribe pattern is well-suited to the IoT domain, where both the number of data-producing sensors as well as the number of data consumers are potentially large and constantly changing [1] [2]. The traditional way of realizing the publish-subscribe communication pattern in IoT is via centralized brokers deployed in the cloud. Well-known technologies used for implementing this approach include MQTT, AMQP, XMPP and ZeroMQ [2].

The centralized solutions have at least two shortcomings. Firstly, they are not scalable in the case of high-volume event streams; the uplink bandwidth cost of a centralized broker grows linearly with the number of subscribers. Secondly, a centralized broker can become a single point of failure, with both technical and business risks present. Technical failures include downtime due to faulty operations or being targeted in a cyber attack, while business risks include vendor lock-in, pricing fluctuations, censorship, data abuse, and even the risk of the service getting shut down altogether.

The limitations of centralized publish-subscribe solutions were recognized early on, and a number of decentralized designs [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] were proposed in the early 2000s' peer-to-peer (P2P) networks research community. During the past five years, decentralized publish-subscribe networks have experienced a renaissance and have been implemented for practical use in IoT and blockchain domains [17] [18] [19].

The decentralized publish-subscribe solutions can, in most cases, offer a constant uplink bandwidth cost for the decentralized message brokers, regardless of the number of subscribers. However, several challenges remain. Firstly, the event delivery latency of a decentralized model is inherently longer than it is in a centralized model, because the events are delivered over a multi-hop application layer multicast. More importantly, the event delivery latency is unpredictable in most cases because of the unpredictable and ever-evolving nature of the peer-to-peer topology. This can make peer-to-peer publish-subscribe networks ill-suited for applications such as warning systems, where predictable latency is important. Secondly, the existing proposals are often vulnerable to various attacks as shown in [20].

In this paper, we present the Streamr Network, a decentralized publish-subscribe network that solves the above-mentioned challenges. In the Streamr Network, each publish-subscribe topic, referred to as a *stream*, forms its separate

P2P overlay topology. Each per-stream topology consists of the nodes tasked with publishing or subscribing messages in the stream. All the nodes participating in a stream contribute upload bandwidth to the stream by forwarding the messages they receive to a fixed number of neighboring nodes in the same topology. As each new subscriber contributes network bandwidth to the system, the topologies can scale without limit. The per-stream P2P topologies are built and maintained by a set of BitTorrent-like trackers [21]. The owners of each stream decide which trackers are to be used for the stream and save the addresses of the trackers in an Ethereum smart contract [22]. The network offers a constant uplink bandwidth cost for publishers and low event delivery latency. The event delivery latency can be predicted by applying Dijkstra's algorithm [23] to a weighted graph that describes the network topology. Moreover, the Streamr Network avoids central points of failure in terms of both technology and management. Compared with previous decentralized solutions based on peer sampling, the Streamr Network is less vulnerable to attacks due to the combined use of blockchain technology and trackers that enforce the intended topologies in the P2P network.

Our real-life experiments, conducted with up to 2048 nodes placed in Amazon data centers around the world, show that, when using a random regular graph as the network topology, the message propagation delays in the Streamr Network scale logarithmically as the number of subscribers in the stream increases, and even in the case of the 2048 nodes network, 99% of messages were delivered globally within 362 ms. The uplink bandwidth cost for the publisher stays constant, and the mean relative delay penalty increases logarithmically, staying under 3.2 in all the experiment runs. Predicting the message propagation delay by applying Dijkstra's algorithm on the network graph, weighted with latencies to each node's neighbors, yields estimates that are within 3.5% accuracy when compared to the measured message delivery latencies.

The remainder of the paper is organized as follows: in Section II, we present background information on previously proposed distributed publish-subscribe systems, and review their design choices. The requirements for the Streamr network are reviewed in Section III. The design of the Streamr Network is presented in Section IV, and its performance is evaluated in Section V. Finally, the paper is concluded in Section VI.

II. BACKGROUND AND RELATED WORK

The publish-subscribe interaction scheme [1] is a popular means of implementing an asynchronous flow of *events* between the *publishers* of events and the *subscribers* who have registered their interest in receiving those events in an event service. Publish-subscribe systems can be roughly divided into three main categories: *content-based*, *topic-based*, and *type-based* systems [1], out of which content-based and topic-based systems are the most common. In content-based systems, subscribers express their interest in events based on the event contents, for example, by specifying event attribute values that the event contents need to match. In topic-based systems, on the other hand, each event is published under a topic, and the subscribers express their interest in receiving certain events by subscribing to topics. In some topic-based systems, topics are hierarchical URL-like strings that allow the use of wildcards when making subscriptions, whereas in others, topics are flat labels with no semantics. In the remaining part of this section we concentrate our analysis on this specific class of publish-subscribe systems that treat topic names as flat labels from the system perspective. [1]

A. TRAFFIC CONFINEMENT

In the most naive strawman-implementation of a decentralized publish-subscribe system presented in [5], the peers would form a global unstructured peer-to-peer mesh, and flood all events to all peers in the network. Out of these events, each peer would then pick the ones it is interested in for further processing. This arrangement would, naturally, lead to a massive waste of bandwidth and high event delivery latencies, as peers would be handling and forwarding events they have no interest in. Thus according to Baldoni et al. [5] it would be beneficial to aim for *traffic confinement* where the events are only delivered to those peers that are interested in them. In [11] the term *noisiness/noiselessness* is used for describing how well a distributed publish-subscribe system succeeds in traffic confinement, and a peer that receives messages it is not interested in is said to suffer from *noise*. Further, in some publications, such as [14] *topic-connectivity* is used as a synonym for traffic confinement. In this article, we use all these three terms interchangeably.

According to [5] traffic confinement should be implemented in peer-to-peer publish-subscribe systems in three steps: *Interest Clustering*, *Inner-Cluster Dissemination* and *Outer-Cluster Routing*. We note that these steps can be also seen as generic strategies that are implemented in some way or another in all topic-based distributed peer-to-peer publish-subscribe systems, regardless of whether traffic confinement is is their explicit goal or not.

1) *Interest Clustering Strategy* defines the means of bringing together, organizing, and maintaining the group of subscribers interested in a certain topic.
2) *Inner-Cluster Dissemination Strategy* defines the means of distributing events within the group of subscribers of a certain topic.
3) *Outer-Cluster Routing Strategy* defines the means of finding the group of subscribers of a certain topic.

In case of the previously-described naive implementation, all the three strategies are implemented by the same simple unstructured overlay network. There is no Interest Clustering, flooding to everybody is used for event dissemination, and

[1] The fact that these systems treat topic names as flat labels from the system perspective does not rule out providing an API layer with hierarchical topic names built on top.

finding any node in the network will suffice for the Outer-Cluster Routing Strategy.

B. DISTRIBUTED TOPIC-BASED PUBLISH-SUBSCRIBE SYSTEMS IN ACADEMIC LITERATURE

Academic proposals for distributed publish-subscribe systems can be roughly divided in two main categories: those based on structured overlay networks (DHTs) and those based on unstructured ones. The former group of proposals includes Bayeux [16], Scribe [6], CAN [13] and Vitis [12], whereas the latter group includes Tera [5], Spidercast [7], PolderCast [14], and Rappel [11].

Early distributed publish-subscribe systems Bayeux [16] and Scribe [6] are both built on top of structured peer-to-peer networks; Bayeux uses the Tapestry DHT whereas Scribe is based on the Pastry DHT network. In both systems, the underlying DHT is used as-is for Outer-Cluster Routing to locate a root node of each topic. As the Interest Clustering Strategy, a per-topic multicast tree starting at the root node is maintained on top of the global DHT topology. Standard DHT routing based on node IDs is used for Inner-Cluster Dissemination, which means that the events need to pass through uninterested intermediate nodes in order to reach their subscribers. Using the terminology of [11], both Bayeux and Scribe are *noisy*, and only partially implement traffic confinement because of the presence of uninterested intermediate nodes, and even the possibility of an uninterested node becoming a topic root in Scribe.

Application-level Multicast using Content-Addressable Networks (CAN) [13] is a DHT-based system that uses a single global CAN DHT for Outer-Cluster Routing to locate bootstrap nodes for multicast groups, but each multicast group is maintained in its own *separate* CAN network. Inner-cluster Dissemination within each multicast group can be implemented either by simple flooding, or by using more elaborate methods that can help to avoid duplicate messages. CAN implements traffic confinement well, except for the fact that nodes in the global CAN DHT network are highly likely to become bootstrap nodes for multicast groups that they do not themselves participate in.

One problem that affects all of the previously presented DHT based systems is the creation of hotspots, where nodes may end up routing large amounts of control data they have no interest in. There are also two other problems with the other DHT-based solutions, except CAN: noisiness, which means the nodes need to route events they are not interested in, and the high relative delay penalty caused by relay-based routing. There are also various DHT-specific attacks [24] that these DHT-based solutions may be susceptible to, if suitable countermeasures are not properly implemented.

Tera, introduced by Baldoni et al. [5], is based on a global unstructured overlay that is used for Outer-Cluster Routing and separate per-topic unstructured overlays for Interest Clustering and Inner-Cluster Dissemination. Outer-Cluster Routing is enabled by advertising the subscriptions of each node to a random subset of nodes in the global overlay, who in turn maintain tables of topic-to-node mappings. Traffic confinement is fully implemented as events are only propagated to the members of corresponding topic-overlays, and there are no root or bootstrap nodes that might act as single points of failure. A potential major problem in the Tera design is the potentially high node degree caused by its reliance on per-topic overlays, in which nodes with large number of subscriptions end up maintaining a large number of connections to other nodes.

Spidercast, presented by Chockler et al. [7], concentrates on Interest Clustering and tries to lower the node degree by tracking the interests of nodes. It uses a single, global, unstructured overlay, in which the neighbor selection is biased, based on the interests of the nodes. Spidercast presumes the existence of a probabilistic distributed membership service similar to [25] or [1], with the help of which each node can construct (at least a partial) *interest view* of other nodes in the network. Each node then chooses neighbors to connect to from this interest view, aiming to keep each topic k-connected (i.e. for each topic, the node is interested in, it maintains connections to k other peers interested in the same topic) while at the same time minimizing the total number of connections to other peers. Through simulation, Spidercast is shown to be noiseless.

Vitis [12] takes a radical approach to the potentially high node degree problem by setting a constant maximum node degree to all the nodes. The result is a Scribe-like overlay where a DHT routing-based relay is used for distributing events to clusters of nodes instead of individual nodes, and the events are flooded within the cluster members using direct relay-free connections. The Vitis system thus has a much smaller noise level much smaller compared to Scribe, but does not eliminate it completely, in order to keep the node degree low.

PolderCast [14] aims at full topic-connectivity (zero noise) while keeping the node degrees at similar level with Spidercast. PolderCast utilizes a distributed peer-sampling service called Cyclon [26] in its Interest Clustering Strategy to get random samples of the whole topology, and a gossip protocol called Vicinity to construct an unstructured overlay network that chooses the neighbors of each node, based on the number of topics any two nodes have in common. A third gossiping protocol called Rings further uses the node samples provided by the Vicinity protocol to generate ring topologies; one ring for each topic. The Inner-Cluster Dissemination Strategy of PolderCast is to push the message both forwards and backwards along the topic ring, and also along the random links within the ring. This strategy results a low number of duplicate messages, and a high dissemination speed measured as number of hops, because of the random links. However, this does not necessarily imply a high dissemination speed measured in milliseconds, because the ring topologies are constructed according to logical node IDs, and network latencies are not taken into account.

Rappel [11] defines itself as a *feed-based* publish-subscribe system in which each feed has exactly one publisher, and a variable number of subscribers. Rappel is similar to PolderCast in the way that it chooses the neighbors in the overlay based on the number of topics a pair of two nodes have in common. However, instead of placing the subscribers of each topic in separate ring topologies like Sub-2-Sub and PolderCast, Rappel builds locality-aware per-topic dissemination trees that aim to minimize event dissemination latency with the help of Vivaldi network coordinates [27]. Rappel achieves full topic-connectivity (zero noise), but according to their own measurements in [11], performs worse than Scribe in terms of event dissemination latency, despite the highly refined locality-aware tree construction algorithm used.

C. SECURITY ISSUES

The previously discussed publish-subscribe systems based on unstructured peer-to-peer networks are all susceptible to the serious threat of eclipse and hub attacks on their peer-sampling-based overlay construction mechanisms [20]. In gossip-based peer-sampling, nodes keep a partial view of the whole network, and periodically exchange parts of their partial views with other nodes. Attacks are possible because the nodes can freely choose the nodes they advertise to the others, and thus byzantine nodes can choose to only advertise their byzantine friends also taking part in the attack.

In an *eclipse attack*, the partial view of a target node is filled with the addresses of byzantine nodes, separating the target node from the rest of the network. The *hub attack* is a larger-scale version of the eclipse attack where the partial views of most of the nodes get filled with the addresses of the byzantine nodes who become *hubs* in the network. If all the hubs, for example, suddenly go offline, they can cause their target nodes to get disconnected from the rest of the network. In [20] it is shown that, utilizing the hub attack, as little as 20 byzantine nodes are enough to bring down a peer-sampling-based network of 1000 nodes in a very short period of time.

There have been numerous attempts at solving the security problems of gossip-based peer sampling, but to the best of our knowledge, all the proposed solutions have serious shortcomings. Some proposed solutions [28] [20] [29] require the use of a certificate authority or some other trusted centralized component, which goes against the idea of peer sampling being a decentralized way of constructing overlay networks. Brahms [30], on the other hand, limits the number of view push operations a client can perform in a certain period of time, using a proof-of-work mechanism. In practical implementations, this would lead to high CPU loads on the network nodes, as the nodes need to run the peer sampling algorithm regularly in order to deal with network churn. GossipSub [19], the protocol planned to be used in the Ethereum 2.0 network, tries to solve the security problems by applying a number of case-by-case mitigation strategies to each attack type. As GossipSub was released very recently, it remains to be seen if its mitigation strategies are extensive enough to make the system secure in practice. The most radical solution to the byzantine nodes problem proposed in Fireflies [31] is moving from partial views to full views of the network. However, according to Johansen et al. [31], the network load caused by keeping the full network view grows linearly with the number of nodes in the networks, and thus the method is only suitable for networks of moderate size.

D. REAL-LIFE NETWORKS THAT RESEMBLE DISTRIBUTED TOPIC-BASED PUBLISH-SUBSCRIBE SYSTEMS

Despite the popularity of the topic in scientific literature, there are only few examples of distributed topic-based publish-subscribe systems that have been deployed in practise. Thus, in this section we will present examples of real-life distributed systems that do not necessarily fit into the exact definition of a topic-based publish-subscribe system, but are similar enough that their designs can be compared to those of the academic systems presented in the previous section. The goal of this comparison is to analyze how the real-life systems differ from their academic counterparts, and discover the reasons for the differences.

1) Matrix

Matrix [17] is a rare example of a widely-deployed general-purpose distributed publish-subscribe system. Matrix is a classical topic-based publish-subscribe system that aims to deliver events published on a topic, or "room" in Matrix terminology, to subscribers who have registered their interest in the specific topic.

The Matrix network consists of *clients* who connect to *Homeservers* that are networked together to build a server-to-server network. The events published in a room are replicated to all Homeservers that have clients that have subscribed to the room in question. The identifiers of the rooms are scoped by DNS domains. Homeservers act as directory servers for their respective DNS domains, and play a role similar to a tracker in BitTorrent.

If Matrix is to be analyzed using the framework defined in the previous section, its Interest Clustering Strategy is to place all the Homeservers participating in a room in per-topic fully-connected mesh with the help of a per-domain tracker. Event flooding in the fully-connected per-topic meshes serves as the Inner-Cluster Dissemination Strategy, and Outer-Cluster Routing is conducted by locating the domain's tracker using DNS.

2) BitTorrent

Even though BitTorrent [21] is not a publish-subscribe system as such, it shares many similarities with publish-subscribe systems. In BitTorrent, users publish torrents, collections of files, that other users can download by joining the overlay of the torrent and downloading pieces of the torrent from other users. In publish-subscribe terms, the ID of the torrent can be seen as a topic, and the pieces of data that the peer share to each other can be seen as the events.

If viewed as a publish-subscribe system, the Interest Clustering Strategy of BitTorrent is building per-topic overlays with the help of a tracker. The per-topic overlays in BitTorrent resemble random graphs with high node-degrees. The Inner-Cluster Dissemination Strategy of BitTorrent is based on "lazy push"; the clients participating in a torrent inform all their single-hop neighbors with HAVE messages about the pieces of the torrent data they already have and, based on the view the nodes construct about piece availability in their neighborhood, they request the pieces from their neighbors according to a piece selection strategy. The design of this Inner-Cluster-Routing Strategy assumes that the size of the HAVE messages is insignificant compared to the size of the data pieces - otherwise the transmission load of the HAVE messages would become prohibitive in a high-degree mesh. The Outer-Cluster Routing Strategy in BitTorrent varies by version. In the traditional BitTorrent [21], the addresses of the trackers responsible for a torrent are hard-coded into the descriptor file, the so-called ".torrent" file of the torrent. These .torrent-files can in turn be found by interested users on websites called BitTorrent indices. In so-called "trackerless" BitTorrent [32] the clients acting as trackers for a torrent are found using a Kademlia DHT. This implies that clients having nothing to do with a torrent in question may end up acting as trackers for the torrent as this is decided by the proximity of the client IDs to the ID of the torrent in the Kademlia identifier space.

3) Blockchain and other Decentralized Consensus Networks

Beside IoT networks, blockchains and other decentralized consensus networks have emerged as a prominent use case for the decentralized publish-subscribe pattern. In the original Bitcoin and Ethereum networks, all full nodes subscribed to a handful of preset topics ("block", "transaction"), whereas the upcoming Ethereum 2.0 network will have hundreds of topics and each node will only subscribe to a subset of them [19].

The Bitcoin network is an unstructured peer-to-peer network of a potentially high node-degree where each public Bitcoin node may establish up to 8 outgoing connections and accept 117 incoming connections [33]. Bitcoin nodes get to know other nodes by fetching lists of node addresses from so-called "DNS seeders" that periodically crawl the network, and through a peer exchange mechanism where peers send lists of other nodes they know to each other in ADDR messages. Nodes store the encountered node addresses on disk, and choose the peers to connect with, according to various criteria reviewed in detail in [33]. The Bitcoin nodes protect themselves against eclipse attacks by grouping known peer addresses into buckets of limited size based on IP prefixes, and by allowing each IP address to be stored only once. Therefore, to successfully eclipse a Bitcoin node, around 8000 IP addresses from various subnets would be needed [33].

The message propagation for the large (500KB) block messages in the Bitcoin network happens utilizing "lazy pull" in a way similar to BitTorrent where the nodes first advertise to their neighbors the availability of blocks using small INV messages, and the neighbors can then request to download the blocks they do not yet have. Significant difference to the HAVE messages of BitTorrent is that the INV messages are only instantly flooded to 25% of the neighbors, and trickled to the remaining 75% of the neighbors at 100ms intervals [34]. Message propagation delays in the Bitcoin network are lengthy and highly variable; in measurements of [35] the median time for a Bitcoin node to receive a newly-created block was 6.5 seconds, mean time was 12.6 seconds, and even after 40 seconds 5% of the nodes still had not received the block.

Ethereum is a blockchain that specializes in programmable on-chain applications called smart contracts [22]. Go Ethereum (Geth), the official reference implementation of Ethereum network, has a default node degree of 25, with 8 outgoing and 17 incoming connections [36]. In Geth, a Kademlia-like DHT is used for peer discovery only. There are no restrictions on which incoming connections are accepted, and the selection of the 8 outgoing connections among the discovered peers also follows a random-like criteria instead of the Kademlia one. Limitations for connections to nodes in a single IP subnet are more relaxed than in Bitcoin, and thus it was shown [36] that two public IP addresses from distinct /24 IP subnets were sufficient for mounting a successful eclipse attack against an Ethereum node. Blocks are propagated in the Ethereum network in two phases, utilizing a gossip-like protocol where the sending node, after initial verification, first pushes the new block to randomly-chosen $\sqrt{d}$ out of its total of d neighbors, and after processing the block completely it advertises the availability of the new block to the remaining $d - \sqrt{d}$ neighbors. Now the neighbors, who have not yet received the block in question, can request the block in pull-based manner [37]. In a recent measurement study [38], block propagation delays in the Ethereum network were found to be low, with 99% of the blocks being propagated in under 317ms between the measurement nodes of the authors placed in 4 locations around the world.

III. REQUIREMENTS

In this section we analyse the requirements for the design of the Streamr Network based on three example scenarios and the goals set in the Streamr crowdfunding whitepaper [39], in which the vision for the Streamr project was first explained.

A. EXAMPLE SCENARIOS

In order to analyze the requirements for our IoT publish-subscribe system, we present three large-scale example scenarios in which the system could be used.

1) Smart City Open Data

A city or a consortium of cities publish real-time sensor readings and public transit data, such as the positions of the public transit vehicles, as open data. A representative example of this kind of service is the digitransit.fi portal [40] that provides high-frequency positioning data of Helsinki region public transit vehicles as real-time MQTT streams.

2) Connected Cars

The smart city is divided into geographic areas, and connected cars moving inside each area publish their sensor readings (e.g. speed, position), notifications and warnings to an events stream covering their particular area. The cars subscribe to the event stream of the area that they are currently in, and, through that stream, receive all events from the other cars in the area.

3) Participatory Environment Sensing

Individual persons living in the smart city publish readings from their own sensors such as thermometers, humidity sensors and air quality sensors. Based on this data a multitude of services can be implemented, such as the weather service envisioned in the "Social Weather Service" scenario of [2] or the air quality monitoring service built in the HOPE project [41].

B. REQUIREMENTS FOR THE STREAMR NETWORK

Using the the motivating scenarios outlined above and the Streamr crowdfunding whitepaper [39] as our sources, we extracted the following 11 main requirements for the design of the Streamr Network.

1) Scalability

According to the goals set in the Streamr crowdfunding whitepaper [39], the network should scale without limit, meaning that the network continue providing acceptable quality of service no matter how many nodes are added to it. Also the load of the publisher should stay constant regardless of the number of subscribers. Keeping the publisher load low and constant is important in all three motivating scenarios listed above. In the "Smart City Open Data" scenario, the number of subscribers may be high, and depends on the popularity of the apps built on top of the data streams. In the "Connected Cars" and "Participatory Environment Sensing" scenarios, the uplink capacity of the connected car or the sensor driver might prevent sending the the events via unicast to all interested subscribers.

2) Decentralization

The the network should not have hotspots or central points of failure. If the network is used by a consortium of cities for distributing open data in the "Smart City Open Data" scenario, the independent cities should be able to keep sharing data using the system even if some of the cities leave the consortium or if the consortium is completely abolished (no single points of failure in terms of management). The cities should be able to keep sharing data independently of technical resources provided by the consortium, and technical problems faced by one city should not affect the service quality of the others (no single points of failure in terms of technology). The same arguments are valid also in the "Connected Cars" scenario, where the messaging system should be independent of car manufacturers or consortia.

3) Low and predictable latency

Each subscriber should receive each event without unnecessary delays. The event propagation delay should stay low in all situations. In addition, the relative delay penalty should be small if compared to having direct unicast connections to the publishers. The "Connected Cars" scenario highlights the importance of this requirement, because sensor readings, and especially warnings from the cars, need to arrive with a predictable delay to be useful.

4) Optimization for small payloads (telemetry, IoT)

The network should be optimized for transmitting large numbers of small messages, in contrast to transmitting small numbers of large messages. To provide an example, the data from the digitransit.fi open data portal mentioned in the "Smart City Open Data" scenario gives a reference on the size and frequency of the event data in an IoT publish-subscribe system. We observed the vehicle position stream of digitransit.fi [2] over a period of 24 hours and measured an average message frequency of 653 messages per second and an average message size of 308 bytes.

5) Bandwidth efficiency

The number of unnecessary messages transmitted in the network should be low. A node should only receive a small number of duplicates of each message, as a high number of duplicates of each event would be inefficient use of the bandwidth of participants in the network. In the "Smart City Open Data" the data subscribers are often mobile web pages and mobile apps such as journey planners. In this case the transmission of unnecessary data also drains the battery of the subscribers.

6) Message completeness

A node should receive all the messages of a stream it is subscribed to with a high probability. The node should be able to detect if it has missed a message, and should be able to fetch the missing message in that instance. This requirement is important in the "Smart City Open Data" scenario, where missing an event might lead to missing a bus for a journey planner user, and it is also important for the "Connected Cars" scenario, where missing a warning from another car could, in the worst case, lead to a traffic accident.

7) Churn tolerance

The network should continue offering a good quality of service even if nodes join and leave the network at a rapid pace. In the "Connected Cars"

[2] mqtt.hsl.fi topic "/hfp/v2/journey/ongoing/"

scenario, the cars are constantly moving around the city from one area to another and connecting to/disconnecting from the streams specific to each area.

8) Zero noise

The nodes should only receive messages from the streams they have subscribed to. The nodes should not need to do any relaying of messages they are not interested in. This requirement is especially important in the "Participatory Environment Sensing" scenario where the individual persons are free to publish their sensor data. It is a beneficial security feature, in the event that a malicious user launches a DDoS attack at the service, or somebody accidentally floods the service by connecting a sensor with a faulty driver the network.

9) Fairness

The network should discourage selfish behavior such as trying to achieve a smaller message latency than the other users. This requirement is not apparent in the three motivating scenarios listed above, but becomes relevant when, for example, transmitting stock market data because receiving data at a lower latency than others might result in monetary gains.

10) Attack resilience

The network should be resilient to all known attacks.

11) Simplicity

The network should be reliable in real-life use. The implementation should be kept simple and easy to debug.

C. REQUIREMENT ANALYSIS

The requirements listed in the previous section set the boundaries for the design of the Streamr Network. In this section, the requirements are analyzed in order to map the design space in which choices can be made for fulfilling each individual requirement.

The two first requirements "(1) Scalability" and "(2) Decentralization" restrict the design space to peer-to-peer networks because client-server designs cannot fulfil either of these requirements. The following nine requirements limit some of the design choices that can be made when implementing the peer-to-peer network.

The requirement "(3) Low and predictable latency" can be addressed with the design of the network topology of the peer-to-peer network. And controlling the topology can be achieved either in a decentralized fashion, as in structured peer-to-peer networks, or in a centralized fashion, as in BitTorrent.

One should also be able to measure and/or estimate the average latency in the network, in order to keep the latencies predictable. Choosing a topology with well-known properties in the academic literature helps in predicting the latency of the network.

The requirements "(4) Optimization for small payloads", "(5) Bandwidth efficiency" and "(6) Message completeness" and "(7) Churn tolerance" go hand-in-hand. "Optimization for small payloads" has the implication that the number of control messages exchanged in the network should be kept low, because there is no significant difference between the size of the control and data messages (both are typically under one MTU). Transmitting a control message increases the bandwidth requirement of the network as much as transmitting a data message, therefore, the number of control messages in the network should be kept as low as possible. This rules out using BitTorrent-like pull-based protocols [42] where nodes first get informed of the availability of new data with the help of a control messages, and only thereafter request the data that they need. The requirement also favors mesh-like topologies over tree-like topologies, such as that of the Plumtree protocol [43], where transmission of control messages is required in order to maintain the tree. In push-based mesh-like systems that use flooding or gossiping for data delivery, it is unavoidable that the nodes will receive duplicates of each data message [10]. However, keeping the number of duplicates low is possible by keeping the node degree (or fanout of each node) low. On the upside, some degree of message duplication makes it easier to achieve message completeness because receiving the same message from multiple peers increases the probability that a node will receive all the data messages in a stream. Message duplication also increases the churn tolerance of the network because each node receives the stream of messages from a number of other nodes simultaneously, and the leaving or crashing of a single node does not cause an interruption in the stream of messages. Duplication also guards against censorship; if a particular node in a topology does not propagate certain messages to other nodes, the messages will still get delivered by other, honest-behaving nodes.

The requirement "(8) Zero noise" implies that there cannot be any relays in the network. This requirement rules out most designs based on structured peer-to-peer networks such as those of Bayeux [16] and Scribe [6]. The requirement "(10) Attack resilience" rules out many of the designs based on peer sampling proposed in the academic literature because they are known to be very vulnerable to eclipse and hub attacks [20]. Some attempts to fix this issue such as Brahms [30] are complex and not proven in practice, and thus are in conflict with requirement "(11) Simplicity". The problem arises in peer sampling because the nodes are free to connect to a large number of other nodes in a short time, and it is possible for an attacker to trick the nodes into connecting to fellow attackers. One way of making this attack more difficult is to prevent the nodes from freely choosing their own neighbors. If the neighbors of each node are decided by trusted entities such as centralized trackers, this also mitigates the problem of selfish nodes trying to optimize their place in the topology, which in turn supports the requirement

"(9) Fairness".

IV. DESIGN

The Streamr Network is a decentralized, topic-based publish-subscribe system. Each publish-subscribe topic, referred to as a *stream* in the Streamr Network, has its own peer-to-peer overlay network that is built and maintained by a set of BitTorrent-like trackers. The Streamr Network operates alongside the Ethereum blockchain, which is used to maintain public and secure registries of streams, trackers, and permissions. The owners of each stream can decide which trackers are to be used for their stream; they can use a set of default trackers maintained in a community-curated Ethereum smart contract, or assign custom trackers by registering the tracker addresses into the stream's smart contract. The trackers can either be operated by the stream owners or a third party trusted by the stream owners. When a node wishes to publish or subscribe to a stream, it first looks up a tracker to contact from the smart contract, and then joins the topology of the stream according to the instructions given by the tracker.

A high-level overview of the Streamr Network is presented in Figure 1. In the figure, a stream owner has registered a set of trackers for his/her stream. A user, "Bob" is interested in the stream and instructs his Streamr node to join the stream. Bob's node first consults the smart contract for a list of trackers, and connects to one of them to find other nodes already in the stream's overlay. According to the instructions given by the tracker, Bob's node joins the stream by connecting to a number of other nodes in the stream's overlay. From this point on, Bob's node will receive all the events published in the stream from these neighbor nodes. Bob's node may also publish its own messages on the stream through these neighbors if the stream owner has given Bob publish permission on the stream.

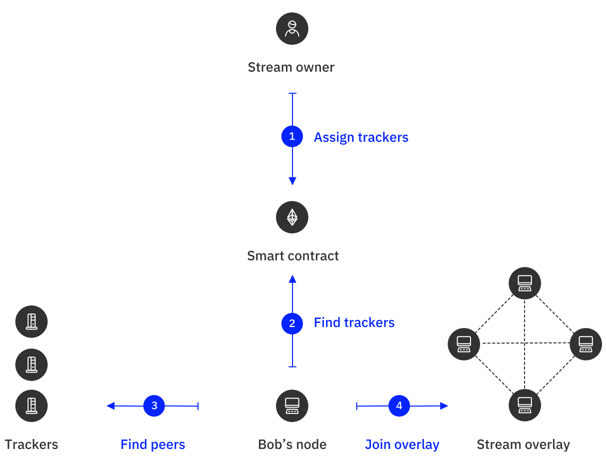

FIGURE 1. High-level overview of the Streamr Network

Tracker mirroring can be used in order to prevent the tracker from becoming a single point of failure for a particular stream in terms of technology, and the independence of streams from each other, as well as the free choice trackers for one's own streams, avoids single points of failure in terms of management.

The tracker controls the stream's topology by instructing the nodes to connect to and disconnect from other nodes, and by instructing the nodes who to accept incoming connections from. Eclipse attacks are difficult to stage because the nodes are not allowed to choose their own neighbors. Because the tracker has control over the connections the nodes can make, it can organize the nodes into different kinds of topologies, and by asking the nodes to ping their neighbors, it can build a map of the stream's topology as a weighted graph. Using this network graph, the tracker can calculate an estimate of the event propagation delay in the current topology by applying Dijkstra's algorithm. This makes it possible to optimize the delay and message redundancy in the network, and to predict the message delivery latency.

A. OVERLAY STRUCTURE

The Streamr Network uses *per-topic random regular graphs [44] of a low node degree* as its default network topology. In such a topology, each node maintains 4 bidirectional Websocket or WebRTC connections to other nodes per topic. Every publish-subscribe topic forms its own separate overlay network, but node-to-node connections are shared between the overlays, and at any point in time there can be a maximum of one open connection between any pair of two nodes. This structure of per-topic overlays with collapsing duplicate links was shown by Chockler et. al. [7] to be able keep the number of connections per node reasonably low in practical situations, while only delivering events to interested nodes.

In the current Streamr Network, events are propagated in the per-topic overlays with simple flooding to all the neighbors of each node, with the exception of the neighbor the event was received from. Gossiping the event to only a random subset of the neighbors is not used, because this would make the event propagation probabilistic and contradict with the requirements "(6) Message completeness" and "(3) Low and predictable latency" defined in the previous Section. Because the node degree of the network is low, we find the benefits of switching from flooding to gossiping to be minimal when compared to the negative impact of adding a deliberately probabilistic component to the message delivery. We acknowledge that there is a fundamental trade-off between bandwidth efficiency on one hand, and simplicity, message completeness, and churn tolerance in the other. In the current design of the Streamr Network we chose to prioritize the latter three design goals at the cost of bandwidth efficiency, and chose flooding in a random regular graph as the event dissemination strategy.

B. OVERLAY MAINTENANCE

The role of the tracker in the Streamr Network is much more active than the role of the Tracker in BitTorrent [45] and the role of the Matrix directory server [17]. In Matrix, the directory server returns to the requesting node the addresses of all the other nodes having the desired channel, and the requesting node connects to them all. In BitTorrent, the tracker, by default, also serves the role of a peer directory from which nodes can find a random subset of the addresses of their peers, and the nodes choose their actual data-exchange partners from this subset by applying a *peer selection strategy*. In the Streamr Network, it is the tracker that executes the peer selection strategy, and sends explicit instructions to nodes to open and close connections to their peers. The default tracker algorithm of the Streamr Network aims at keeping the topology of each per-topic overlay as close as possible to a random regular graph with a low node degree and a low network diameter. The tracker algorithm is inspired by the Steger and Wormald [46] algorithm, shown below, for generating a random regular graph of n nodes with node degree d:

1) Start with nd points 1, 2, . . . , nd (nd even) in n groups. Put U = 1, 2, . . . , nd. (U denotes the set of unpaired points.)
2) Repeat the following until no suitable pair can be found. Choose two random points i and j in U, and, if they are suitable, pair i with j and delete i and j from U.
3) Create a graph G with edge from vertex r to vertex s if and only if there is a pair containing points in the rth and sth groups. If G is d-regular, output it, otherwise return to step (1).

In the Steger and Wormald algorithm the points i and j are considered suitable if i and j do not belong to the same group and there is no existing pairing between the groups that i and j belong to. If we interpret this algorithm in terms of peer-to-peer networks, the groups correspond to the nodes in the network, each node having d connection slots. In this interpretation, the suitability condition means that no node is allowed to connect to itself and there can be at most one connection between any given pair of nodes. However, The Steger and Wormald algorithm is aimed at constructing a random d-regular graph out of a static group of n nodes and does not address the actual situation in peer-to-peer networks where n is constantly changing as nodes join and leave the network. Taking this dynamism into account, we designed the following tracker algorithm that is executed by the tracker every time a node joins or leaves the network:

1) Let V denote the set of nodes with unused connection slots in the system at the time the algorithm is executed.
2) Repeat the following until no suitable pair can be found: choose two random nodes a and b in V, and if they are suitable, connect a with b and delete a and b from V if they do not have unused slots anymore.
3) Delete all nodes from V that have less than 2 unused slots.
4) Repeat the following until there are no nodes left in V: for node $v \in V$ pick uniformly at random from the network nodes c and d that are connected to each other and are suitable for connecting to v, and disconnect c from d. Now connect v to both c and d, and delete v from V if v has less than 2 unused slots left.

In addition to the Steger and Wormald algorithm, the Streamr tracker algorithm resembles the distributed algorithms used for network maintenance of d-regular random graphs in the SWAN system [47]. The SWAN algorithm consists of three distinct strategies that are executed when a node joins the network, a node disconnects from the network gracefully, and when a node disconnects from the network ungracefully without a warning:

1) When a new node joins the network, it initiates d/2 random walks to choose d/2 disjoint connections in the network at random. The new node interposes itself on each chosen connection, placing itself in between the two nodes the chosen connection originally connected.
2) When a node disconnects the network gracefully, it instructs its d neighbors to connect to each other directly (d is always assumed to be even). Effectively the node de-interposes itself from between its neighbors, making this strategy the exact reverse of the strategy 1).
3) When a node disconnects from the network disgracefully without a warning, its neighbors will notice this and broadcast a request for new connections through the network. With the help of these connection requests, the nodes with less than d connections can pair up and make the network complete.

We can notice that strategy 1) of SWAN corresponds to stage 4) of the Streamr tracker algorithm where both choose disjoint connections in the network at random, and interpose the node with less than d connections between the two nodes originally connected together. Strategy 3) of SWAN, in turn, corresponds to stage 2) of the Streamr tracker algorithm, where nodes with less than d connections are paired together. Even though there is no equivalent to SWAN's strategy 2) in the Streamr tracker algorithm, we argue that this is insignificant to the practical properties of the resulting network and that the theoretical results on SWAN-like networks found in the literature [48] can also be applied to the Streamr Network.

C. NETWORK PERFORMANCE ESTIMATION

One of the requirements for the design of the Streamr Network listed in Section III was the predictability of latency. We addressed this requirement by introducing the capability of conducting near-real-time network performance estimation to the tracker. At all times, the Streamr tracker keeps an up-to-date graph of each per-stream topology it controls. This graph contains information about the one-way delays of the connections in the topology, and the delay values are kept up-to-date via status reports periodically sent by each node to the tracker. The reports include the round-trip-delays the node constantly measures to its d neighbors. Using this graph, the tracker can calculate an estimate of the end-to-end multi-hop delays in the topology at any point in time, using the standard Dijkstra's algorithm. Any application interested in the delay estimates for any particular stream can request this data from the tracker of the stream, to gain knowledge of metrics such as estimated minimum, maximum and mean message propagation delays in the stream.

V. THEORETICAL ANALYSIS AND EXPERIMENTAL EVALUATION

We evaluate the design of the Streamr Network using theoretical analysis, simulations and live experiments. Simulations are used to show that the Streamr tracker algorithm produces topologies that are similar to random d-regular graphs. Theoretical analysis is then used to demonstrate what kind of trade-offs happen due to the fact that Streamr topologies are similar to random d-regular graphs, and all the implications that might have. Finally, the results of live experiments, conducted by placing up to 2048 Streamr nodes in Amazon data centers around the world, are presented to demonstrate the real-life performance and scalability characteristics of the Streamr Network and to show the accuracy of the network performance estimation method introduced in section IV-C.

A. METRICS AND DEFINITIONS

We use the following metrics and definitions when evaluating the performance of the Streamr Network:

Node degree is the number of connections each node has in a stream's topology.

Network diameter (expressed and evaluated as number of hops) is the longest of the shortest paths between the pairs of nodes in the topology.

Shortest path delay expressed in milliseconds is the lowest one-way delay between a pair of nodes in the topology.

Round-trip time (RTT) expressed in milliseconds is the the underlay round-trip time between two nodes in the topology.

Relative delay penalty (RDP) is the ratio between the one-way delay between two nodes in the overlay network and the one-way unicast delay between the same two nodes in the underlay network (half of the RTT). When evaluating the performance of network topologies, we use the metric *average relative delay penalty*, defined as the ratio between the average node-to-node delay in the overlay network and the average node-to-node unicast delay in the underlay network. As pointed out by [49], using average delays in the calculation instead of node-to-node delays helps to avoid the problems caused by node-pairs with small delays seen in [50].

Flooding time (expressed in milliseconds) is the time it takes before all the nodes in the network have received a published message. In an ideal implementation, the flooding time equals the maximum fastest path delay of the network.

Message redundancy (expressed as a count of messages) is the average number of times each node receives each unique message.

B. THEORETICAL ANALYSIS AND SIMULATION

In this section we demonstrate that Streamr Network topologies resemble random d-regular graphs, and we analytically explore the practical implications of this.

1) Randomness and d-regularity of the Streamr topologies

The d-regularity of Streamr Network topologies depends on the tracker algorithm presented in section IV-B. If properly implemented, the tracker algorithm should only generate network topologies where the node degree is at most d for all nodes. Further, with the exception of topologies where the number of nodes is less than or equal to d, the node degree should be exactly d for all even values of d.

The randomness of the generated Streamr Network topologies is difficult to prove formally. Thus we resorted to simulating the tracker algorithm and comparing the properties of the resulting topologies with those of a random d-regular graph of the same size and node degree. In the Streamr Network, there can be at most one connection between each pair of nodes, and nodes are not allowed to connect to themselves. With these assumptions, in a network of n nodes it is possible to establish a maximum of $\frac{n^2}{2} - \frac{n}{2} = \frac{n^2-n}{2}$ distinct connections between the nodes. In every d-regular network of n nodes with node degree d, where d is even and $n > d$, there are $n * \frac{d}{2}$ connections. Thus the probability of a connection existing between nodes v_i and v_j where $i, j \in [0...n]$, $i \neq j$ is

$$P(\mathrm{v}_i \text{ is connected to } \mathrm{v}_j) = \frac{n * \frac{d}{2}}{\frac{n^2-n}{2}} = \frac{d}{n-1}. \tag{1}$$

For example, in a random d-regular network with 1000 nodes and node degree 4 we get the probability for connection existing between any given pair of nodes in the network to be $\frac{4}{1000-1} = \frac{4}{999} \approx 0.0040040$.

In order to study the randomness of the topologies generated by the Streamr tracker, we extracted the tracker algorithm from the Streamr Network codebase into a separate module, and simulated the construction of a topology of $n = 1000$ nodes and node degree $d = 4$. The nodes were given distinct identifiers in the range of $[0...999]$ and they joined the topology in order based on their identifier. We repeated this simulation 1000 times, and finally calculated how many times there was a connection between each distinct pair of nodes in the 1000 topologies. The results are shown in the connectivity counts matrix in Figure 2. Each pixel in the graph represents a connection from node v_i to v_j in the 1000 generated topologies. The darker the pixel is, the more times the connection existed in the topologies. Note that the matrix is mirrored along the diagonal as the connections are two-way.

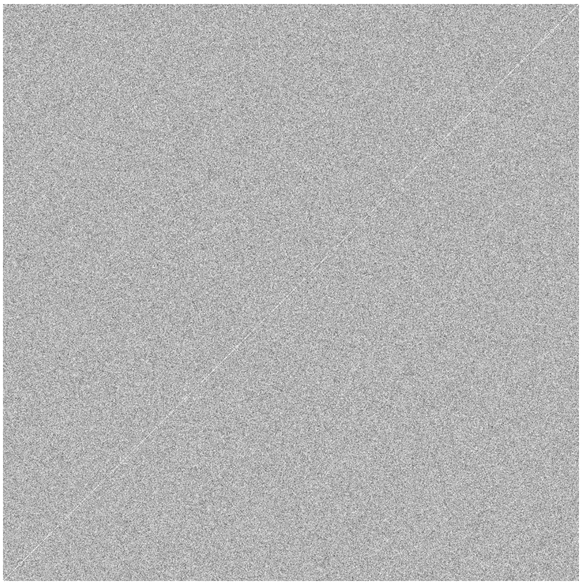

FIGURE 2. The pairwise counts of neighbor assignments for 1000 nodes with node degree 4 over 1000 rounds. Each row and column corresponds to a node, and both are ordered according to join order of nodes. A pixel at (i, j) is darkened according to how many times nodes v_i and v_j were made neighbors. The matrix is symmetric, and the diagonal is white ($(i, i) = 0$) because a node cannot be assigned to be its own neighbor.

We can observe that the connectivity counts matrix presented in Figure 2 resembles white noise, which can be seen as an indication of randomness. To analyze the randomness further, we calculated the average probability of a distinct pair of nodes being connected in the topologies by dividing the counts by the number of repetitions. The calculated average probability for a pair of nodes being connected in the simulations was 0.0040040, which matches the probability of 0.0040040 calculated using Equation 1 for a perfect random d-regular graph. The distribution histogram of the probabilities of a distinct pair of nodes being connected in the simulation data is shown in Figure 3.

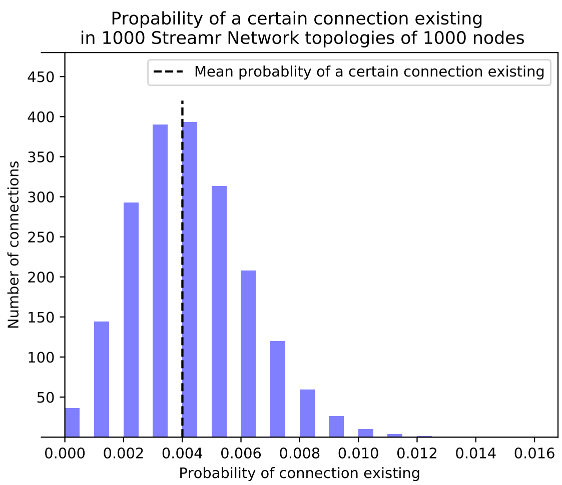

FIGURE 3. Histogram of the probabilities of a distinct pair of nodes being connected in the simulation data

We can observe from the figure that the probabilities of a distinct pair of nodes being connected in the simulation data are distributed around the mean as would be expected of a random variable [3]. We can thus safely conclude that the topologies generated by the Streamr tracker algorithm closely *resemble* a random d-regular graph, even though we leave the exact proof of this for future work.

2) Analysis of practical design trade-offs

Having shown in the previous subsection that the Streamr Network topologies resemble random d-regular graphs, we can now analyze the practical design trade-offs of the network based on what is known about random d-regular graphs in the literature.

One of the most used metrics in the literature for evaluating network topologies is the *network diameter*. It is defined as the maximum hops on the shortest path between any pair of nodes in the network [44]. According to analytical results [44], the network diameter $diam(G)$ for a random regular graph G with n nodes of node degree d satisfies (asymptotically almost surely) the condition

$$1 + [log_{d-1}\ n] + \left[log_{d-1}\left(\frac{(d-2)}{6d} log\ n\right)\right] \leq \\ diam(G) \leq 1 + [log_{d-1}((2+\epsilon)\ d\ n\ log\ n)]\,. \tag{2}$$

Figure 4 displays the upper bound of the network diameter defined in (2) as function of number of nodes n in the network with various node degrees d.

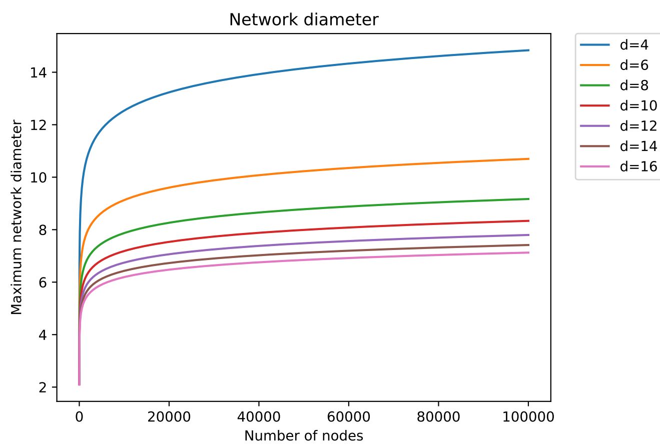

FIGURE 4. The upper bound of network diameter according to Equation 2

It can be observed in Figure 4 that the upper bound of the network diameter increases logarithmically as the number of nodes increases. Another key observation from the figure is that the upper bound of the network diameter decreases as the node degree d increases. In order to make this phenomenon more visible, in Figure 5 we plotted the upper bound of the

[3] A subtle long tail can be observed, which could indicate that nodes arriving first have a very slightly higher probability of being connected to each other

network diameter at various node degrees while keeping the number of nodes constant $n = 100000$.

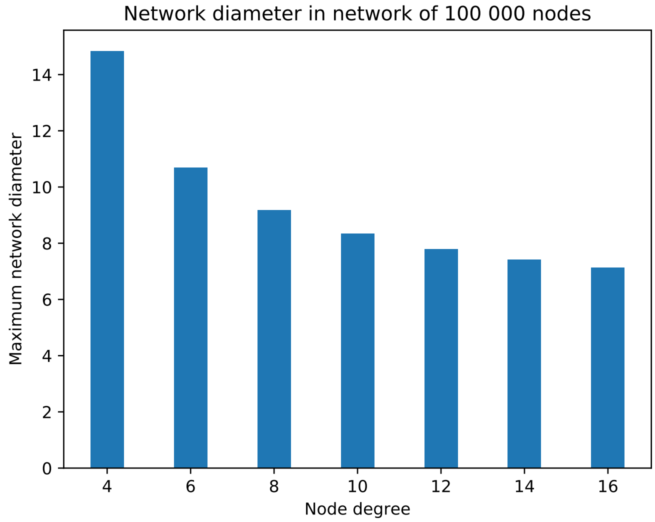

FIGURE 5. The upper bound of network diameter at various node degrees according to Equation 2

The upper bound of the network diameter decreases at a decreasing rate, as the node dimension d approaches its maximum value, $d = n - 1$, of a fully-connected network.

Message redundancy is a useful metric in evaluating how the Streamr Network meets its requirement "(5) Bandwidth efficiency". Message redundancy in flooding in random graphs was analyzed in [10], and the upper limit for the number of unavoidable duplicate messages in flooding a message was given as

$$N\left(\frac{\overline{d}}{2} - 1\right) + 1 \tag{3}$$

where N is the number of nodes in the network, and $\overline{d}$ is the average node degree. However, the problem of this formula is that it assumes a handshake is made prior to sending each message to ensure that each link is used exactly once. Such a handshaking protocol is not feasible in a system such as the Streamr Network that aims at low latency and is optimized for small payloads, as the handshakes would at increase the delay with at least one RTT, and the handshaking messages would not be considerably smaller than the actual message payloads. We thus need to derive a more realistic formula for the number of duplicates in the Streamr Network.

When a message is flooded in a Streamr Network topology on $N nodes$, the publishing node first sends it to its d neighbors. As each of the $N - 1$ non-publisher nodes receives a message for the first time, it forwards the message to $d - 1$ of its neighbors (all neighbors except the one it received the message from) in a quick, atomic operation that cannot be interrupted.

Thus the total number of messages sent during the flooding of a single message in the Streamr Network is $(N - 1)(d - 1) + d$. In an optimal case with no duplicates $N - 1$ messages would get sent, and thus $(N-1)(d-1)+d-(N-1) = N(d-2) + 2$ of the sent messages are duplicates. In conclusion, we can express the average number of duplicates sent in the flooding of a single valid message in the Streamr Network per node as

$$\frac{N(d-2)+2}{N} = d - 2 + \frac{2}{N} \approx d - 2. \tag{4}$$

The number of duplicates received per each valid message by each non-publisher [4] node as function of node degree is plotted in Figure 6.

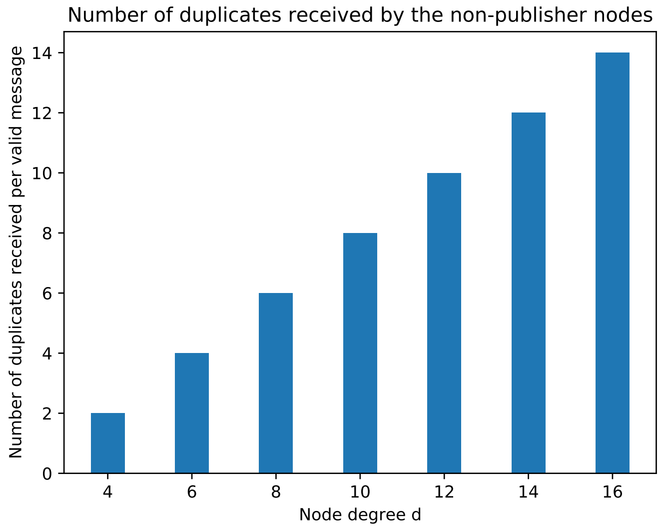

FIGURE 6. Number of duplicate messages received by the non-publisher nodes per each valid message as function of node degree

For example, if 4 messages per second are published in a Streamr Network topology with node degree $d = 6$, each node in the network will receive $4 + (6 - 2) = 8$ messages per second.

The number of duplicates per valid message received translates directly into downlink bandwidth demand of the non-publisher nodes. If node degree of 4 is used, the downlink bandwidth demand of the nodes will be 3 times the stream bandwidth, with node degree 6, the bandwidth demand will be 5 times the stream bandwidth and so on. The node downlink bandwidth demand factor as function of the node degree can be formally expressed based on Equation 4 as

$$1 + (d - 2) = d - 1. \tag{5}$$

.

The uplink bandwidth demand of factor of the the non-publisher nodes is also $d - 1$, as each non-publisher node forwards each message to $d - 1$ neighbors. For the publisher node, the uplink bandwidth demand factor is exactly d, because it sends each message to all of its d neighbors.

[4] the publisher node receives no messages

As is evident from Figures 5 and 6 the *node degree* is a configuration parameter that allows one to trade smaller network diameter (lower latency) for greater redundancy (higher bandwidth consumption). Choosing the right node degree for each per-topic overlay thus becomes an important design decision, where a balance between, on the one hand, the requirement "(5) Bandwidth efficiency" and on the other hand, the requirements "(3) Low and predictable latency", "(7) Churn tolerance" and "(10) Attack resilience" needs to be found. With higher node degrees yielding diminishing returns in network diameter, for the Streamr Network, we chose the node degree 4 as the best default value for most use cases, although stream owners could customize this setting on a per-stream basis to achieve their preferred trade-off.

It should also be noted that the network diameter, measured in hops, is not the same as delay measured in time. In the real world, the one-hop delays may differ significantly, as a connected pair of nodes might be in the same data center and have a low unicast delay between them, while another pair of connected nodes could be on opposite sides of the world. For practical applications, only the delay measured in milliseconds is relevant, not the number of hops required for the data to reach its destinations. For this reason, in case of a publish-subscribe network intended to operate globally, it is useful to measure delays in geographically distributed real-world setups with heterogeneous delays between nodes. In the next section, we will explore such an experiment.

C. AMAZON EXPERIMENTS

In order to measure the real-life performance of the Streamr Network, we conducted a series of experiments on Streamr Network nodes deployed to 16 Amazon Web Services (AWS) regions around the world. Our goal was to study the scalability of the Streamr Network by observing how the key metrics, message propagation delay and relative delay penalty evolve as the number of nodes in the network increases. Additionally, we aimed to evaluate the accuracy of our network performance estimation method presented in Section IV-C.

1) Experimental setup

The experiments were conducted using a variable number of Streamr Network nodes deployed at 16 distinct Amazon AWS regions listed in Table 1, and one Streamr tracker, configured to construct topologies of node degree 4, was deployed at the eu-west-1 AWS region (Ireland). As the number of nodes was varied in the experiments, every participating AWS region always had an equal number of nodes. Each Streamr node was run in its own virtual machine of type "t3.small" in an EC2 auto scaling group (ASG) that was used for starting the right number of virtual machines for each experiment run. Systemd was used inside each virtual machine to automatically start and stop the Streamr node whenever the virtual machine started or stopped. A simple HTTP endpoint was added to the Streamr nodes to allow for remote control during the experiments. The endpoint was used to instruct the nodes to subscribe and unsubscribe to streams, to publish events to streams, to measure round-trip delays, to restart the nodes and to allow fetching of the experiment logs for analysis. The tracker was fitted with a similar HTTP endpoint to allow restarts and fetching of the state of the overlay network topology.

Prior to starting the experiments, we measured using the ping command the average underlay round-trip times (RTTs) between a total of 1024 virtual machines, with 64 virtual machines placed in each of the 16 Amazon AWS regions. Each virtual machine pinged the 1023 other virtual machines five times. Assuming symmetrical links, we calculated the average one-way delay between the regions by dividing the RTTs by two. The results of this measurement are listed in Table 1.

From Table 1 we can observe that the one-way delays between two nodes in the same region were between 0.07 and 0.15 milliseconds, with a mean of 0.12 and a median of 0.095 ms. In the actual experiments, the measurement precision for one-way delays was limited to 1 ms, meaning that node-to-node delays within a single region would appear as zeros in the measurement results if no corrections were made. For this reason we used 0.1 ms as an approximation for all one-way delays between two nodes in the same region in the actual experiments.

2) Conducting the experiments

In the Amazon experiments we aimed to determine the overlay node-to-node message propagation delays between all pairs of nodes in each experiment.

To this end, we started the targeted number of virtual machines on the AWS regions, and once the nodes running inside the virtual machines had started, we instructed them over the node's HTTP endpoints to subscribe to the default stream of the experiment. The nodes joined the stream with the help of the tracker, and after all the nodes had joined the stream, we instructed each node to publish 2 messages to the stream with a 3-second interval between the messages. The publishing instructions were sent to the HTTP endpoints of the nodes in series, which resulted in an average rate of 13.6 messages being published in the network per second. The publishing rate was kept this low in order to prevent network bandwidth or the processing capacity limits of the nodes from affecting the latency measurements [5].

The clocks of the virtual machines were kept in sync using NTP. The publisher and the subscribers appended their node ID and a timestamp to all the messages they encountered in the experiments. The nodes logged every message that they received, and the message propagation delay of each message could be easily calculated with the help of the timestamps attached to the messages. Because there is message duplication by design in the network, only the copy of each message that was first to arrive was taken into account when calculating

[5]The maximum throughput of a node is mainly determined by the available hardware and network connectivity.

the message propagation delays. Additionally, we logged the number of duplicates received for each message.

The experiment was run with network sizes of 32, 64, 128, 256, 512, 1024 and 2048 nodes, which was realized by running 2, 4, 8, 16, 32, 64 and 128 virtual machines (VMs) per AWS region. For each network size, the experiment was repeated 10 times using the same VMs, only restarting the nodes and the tracker using the HTTP endpoints in between. We noticed from the VM logs that in the first repetition of the experiment for each network size, Amazon's CPU throttling had kicked in, introducing artifacts to the measurement results. CPU throttling was not detected during the following 9 repetitions, and thus we discarded the first repetition as warm-up and collected artifact-free measurement results from the other 9 repetitions of the experiment for each network size. The nodes joined the stream anew for each repetition, and thus the tracker generated 63 network topologies in total during the 9 measured repetitions for all 7 different network sizes. At the end of each experiment repetition, the network topology graph was downloaded from the tracker, and the message arrival metrics were downloaded from the nodes, both saved for further analysis.

3) Message propagation delay

As described above, the clocks of the nodes were kept in sync using NTP. Each published message was timestamped by the publishing node, and the receiving nodes logged the arrival time of the first copy of each message. Based on the logs we calculated the *message propagation delay* for each message sent in the experiments.

Figure 7 shows the CDF of the message propagation delays in the experiments with 32, 64, 128, 256, 512, 1024 and 2048 nodes, respectively. The experiment was repeated 9 times for each network size with the tracker generating a new topology for each repetition.

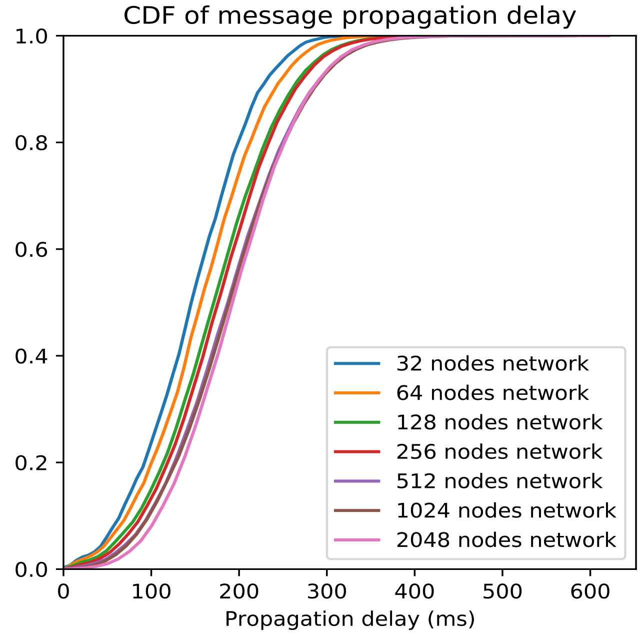

FIGURE 7. Message propagation delays in the Amazon experiments

We can observe from figure 7 that the message delivery delay increases as a function of network size as expected, and that even in the case of the 2048 nodes network, 99% of messages were delivered globally within 362 ms.

In order to examine the *scalability* of the network more closely, we further plotted in Figure 8 the mean message propagation delay as a function of the number of nodes in each experiment. As the experiment was repeated 9 times for each network size with a freshly-generated topology for each repetition, we also show as a shadowed area around the mean the minimum and maximum of the average delays of the different repetitions.

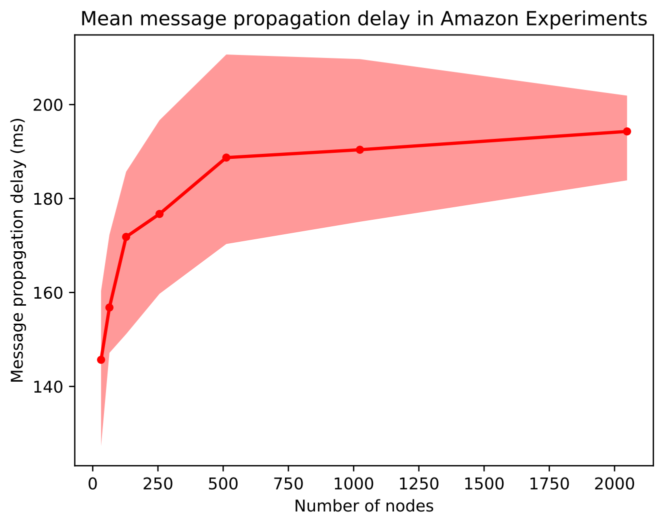

FIGURE 8. Average message propagation delays in the Amazon experiments

We can observe from Figure 8 that the mean message

TABLE 1. Mean one-way delays between nodes in different regions (ms)

	eu-central-1	eu-west-1	eu-west-2	eu-west-3	eu-north-1	us-east-1	us-east-2	us-west-1	us-west-2	ca-central-1	ap-south-1	ap-northeast-2	ap-southeast-1	ap-southeast-2	ap-northeast-1	sa-east-1
eu-central-1	0.08	12.59	7.54	4.47	10.88	42.93	48.35	72.00	82.26	49.97	54.22	132.85	83.90	140.63	126.86	102.74
eu-west-1	11.49	0.13	5.94	9.08	18.69	35.46	47.66	69.64	67.02	39.05	61.15	122.02	90.11	126.88	111.99	92.27
eu-west-2	6.56	4.86	0.07	3.82	12.67	37.95	42.73	67.90	65.70	44.02	54.30	124.00	89.26	136.15	104.59	97.34
eu-west-3	4.50	8.91	3.59	0.07	14.38	39.18	44.51	68.60	78.24	46.07	52.27	135.19	83.26	136.62	122.24	99.28
eu-north-1	10.42	17.43	12.42	14.02	0.07	51.10	57.97	82.06	84.86	56.20	65.91	142.86	98.90	147.24	131.77	109.89
us-east-1	42.92	35.15	37.90	39.27	52.29	0.46	5.67	29.99	37.46	7.60	90.78	92.47	119.30	98.54	80.27	60.37
us-east-2	48.19	47.71	43.52	44.66	58.11	5.71	0.13	25.32	34.53	12.81	95.13	92.30	112.00	93.70	79.11	65.69
us-west-1	72.37	72.21	67.68	68.72	82.09	30.16	25.80	0.10	10.78	38.51	115.94	68.02	89.35	68.28	55.97	95.03
us-west-2	78.10	61.46	69.01	76.24	83.93	39.99	35.05	10.21	0.14	33.44	114.57	61.17	86.59	68.72	49.72	90.04
ca-central-1	49.75	39.07	44.00	46.92	56.49	7.05	12.81	38.63	32.87	0.08	97.24	91.35	112.87	97.98	77.61	62.62
ap-south-1	54.27	59.42	56.24	53.26	64.63	91.46	95.20	120.20	109.85	97.84	0.11	77.85	26.03	69.67	65.77	151.04
ap-northeast-2	134.19	119.23	123.99	135.35	142.88	93.07	92.08	67.09	61.21	91.61	79.62	0.11	51.44	73.50	16.75	148.06
ap-southeast-1	88.68	88.85	84.70	83.30	97.66	114.67	114.57	87.74	81.18	112.62	30.73	51.43	0.09	45.08	42.18	169.35
ap-southeast-2	141.22	128.98	136.56	137.16	147.65	98.11	93.66	68.06	68.97	97.82	69.60	73.50	45.09	0.08	52.74	154.67
ap-northeast-1	125.07	105.30	104.35	122.96	128.86	80.23	79.29	55.95	51.05	78.04	68.14	16.26	39.98	52.71	0.08	134.58
sa-east-1	103.01	91.93	96.37	99.29	109.92	58.02	64.88	94.69	89.59	62.63	151.45	148.29	172.10	154.62	134.45	0.15

propagation delay increases sub-logarithmically as the number of nodes in the network increases, and we can thus conclude that the scalability of the Streamr Network meets the requirements set in section III also in a real-life setting. The data presented Figure 8 is shown in numerical form in Table 2.

TABLE 2. Average message propagation delays in the Amazon experiments

Number of nodes	Min of average delays (ms)	Avg of average delays (ms)	Max of average delays (ms)	Min-Max difference (ms)	Difference %
32	127	146	160	33	26.0
64	147	157	172	25	17.1
128	151	172	186	35	22.9
256	160	177	197	37	23.1
512	170	189	211	40	23.7
1024	175	190	210	35	19.8
2048	184	194	202	18	9.8

The minimumum, maximum, and average delays for each number of nodes listed in Table 2 show the variation in performance between the topologies of the different experiment repetitions. The difference in measured average delays is at its highest when the number of nodes is low, and gets smaller as the number of nodes increases. The variation also gives us a rough indication of the optimization potential in the space of regular graphs, potentially available by utilizing a better-than-random topology construction heuristic [6]. The largest observed difference between the fastest and slowest topology for any network size was 26%, providing a ballpark figure of how much the worst-case delays could be improved by finding an optimal topology.

4) Relative delay penalty

To make the experimental results on the performance of the Streamr Network comparable to the performance of other similar networks known from the academic literature, we calculated the average relative delay penalty (RDP) from our Amazon experiments. We use the definition of average relative delay penalty given in Section V-A where the RDP of the network is calculated by dividing the average one-way delay between the nodes in the overlay network by the average one-way delay between the nodes in the underlay network. In our calculation we used the one-way delays listed in Table 1 as the underlay delays, and the message propagation delays logged by the nodes, as explained in Section , as the overlay delays. The RDP results are plotted in Figure 9. The solid line marks the average RDP of the 9 experiment repetitions. The shaded area around the average line displays the minimum and and maximum average RDP of the repetitions.

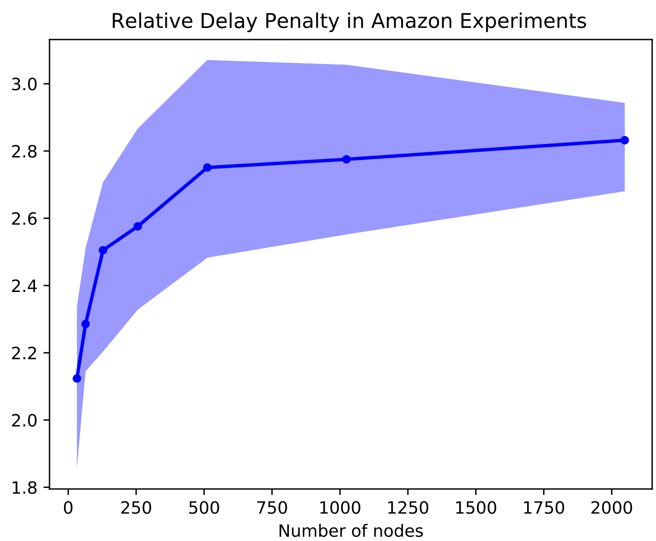

FIGURE 9. Average relative delay penalty in the Amazon experiments. Minimum and maximum average delay penalties of 9 experiment repetitions displayed as shadowed area.

We can observe from the figure that the average RDP stays under 3 in most of the repetitions, and that the difference in average RDP between the fastest and slowest topology gets smaller as the number of nodes in the network increases.

To interpret the RDP result from a practical point of view, broadcasting messages over the Streamr Network to thousands of recipients has a latency of up to 3 times the latency of a direct connection. On the other hand, over the Streamr Network a publishing node can deliver messages to *any* number of recipients by only ever sending out 4 copies of the

[6]Examples of such approaches could be leveraging location or latency information in the topology construction.

message. As discussed earlier in this paper, a large number of direct connections is hardly scalable and comes with the technical and business risks of centralized solutions, making this latency trade-off potentially worthwhile for many practical applications seeking scalability, fault tolerance, and other aspects of risk reduced via decentralization.

5) Number of duplicates

To validate the *number of duplicates* prediction presented in Equation 4 in section V, we logged the number of duplicate messages received by nodes for each individual message. The average number of duplicates in each experiment repetition was found to be in the range of 1.95 to 2.05, matching the expected value of 2 for node degree 4.

6) Performance estimation accuracy

In order to investigate the accuracy of the network performance estimation method presented in Section IV-C, we applied the method for estimating the average latency to each of the 63 network topologies in the Amazon experiment. The network topologies were downloaded from the tracker, and pre-measured delays listed in Table 1 were inserted as weights into the topology graphs. Because the underlay delay measurements were aggregated to a data center granularity, we used the same measured inter-data center delay value for all connections between each pair of data centers. From these weighted topology graphs we calculated the estimated average node-to-node overlay delay in each of the 63 topologies using Dijkstra's algorithm. We further grouped the results by the number of nodes in the topologies, and calculated the average of the estimated average node-to-node delays for each number of nodes. The results are listed in Table 3 and plotted in Figure 10 along with the measured average node-to-node overlay delays from the Amazon experiments.

TABLE 3. Estimation accuracy

Number of nodes	Average of estimated average delays (ms)	Average of measured average delays (ms)	Absolute difference (ms)	Difference %
32	137	146	9	6.3
64	150	157	6	4.3
128	168	172	4	2.6
256	172	177	5	2.7
512	184	189	5	2.7
1024	186	190	5	2.6
2048	189	194	6	3.0

s

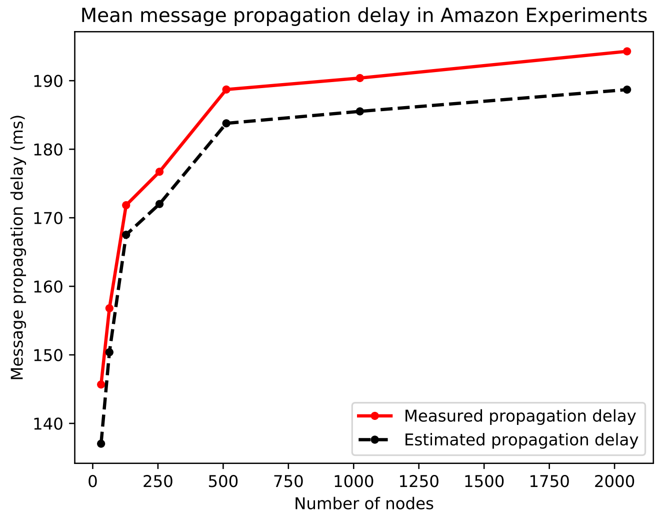

FIGURE 10. Estimated and measured mean message propagation delays in the Amazon experiments.

Table 3 and Figure 10 show that the average of the estimated average delays is 2.6 to 6.3% lower than its measured counterpart for all the investigated topology sizes. This is an expected result, because the Dijkstra's algorithm calculates the fastest paths between the pairs of nodes without taking into account processing delays at the nodes. The delays calculated using Dijkstra's algorithm can thus be seen as a theoretical lower bound for the delays in any given topology, and considering this, the delays in the Streamr Network are not far away from the lower bound.

We plotted the CDF of the absolute percent errors of the 63 estimations into Figure 11.

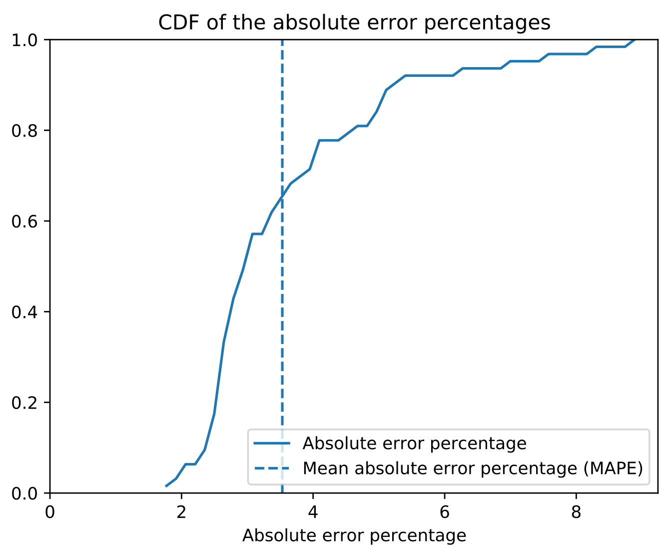

FIGURE 11. Delay estimation error

The mean absolute percentage error (MAPE) calculated from the 63 estimations was 3.5%. The error is surprisingly low, considering that the estimations were calculated using inter-data center delays instead of node-to-node ones, and that the inter-data center delays had been measured several hours before running the actual Amazon experiments.

VI. CONCLUSION

In this paper we presented and evaluated the Streamr Network, a decentralized publish-subscribe system, already in production, that offers low and predictable message propagation delays by enforcing the intended network topology with the help of trackers. An experimental evaluation of the Streamr Network was conducted by running up to 2048 Streamr Network nodes placed in Amazon data centers around the world. The experimental results show that the message propagation delays in the Streamr Network scale logarithmically as a function of the number of nodes, and that the delays can be estimated using Dijkstra's algorithm with mean absolute percentage error (MAPE) of 3.5%.

REFERENCES

[1] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[2] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting iot platform requirements with open pub/sub solutions," *Annals of Telecommunications*, vol. 72, 07 2016.

[3] V. Bourassa and F. Holt, "Swan: Small-world wide area networks," in *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, ser. SSGRR '03, 01 2003.

[4] S. Baehni, P. T. Eugster, and R. Guerraoui, "Data-aware multicast," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 233–. [Online]. Available: http://dl.acm.org/citation.cfm?id=1009382.1009738

[5] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "Tera: Topic-based event routing for peer-to-peer architectures," in *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, ser. DEBS '07. New York, NY, USA: ACM, 2007, pp. 2–13. [Online]. Available: http://doi.acm.org/10.1145/1266894.1266898

[6] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I T Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *Selected Areas in Communications, IEEE Journal on*, vol. 20, pp. 1489 – 1499, 11 2002.

[7] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication," in *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, ser. DEBS '07. New York, NY, USA: ACM, 2007, pp. 14–25. [Online]. Available: http://doi.acm.org/10.1145/1266894.1266899

[8] S. Girdzijauskas, G. Chockler, Y. Vigfusson, Y. Tock, and R. Melamed, "Magnet: Practical subscription clustering for internet-scale publish/subscribe," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '10. New York, NY, USA: ACM, 2010, pp. 172–183. [Online]. Available: http://doi.acm.org/10.1145/1827418.1827456

[9] M. Matos, A. Nunes, R. Oliveira, and J. Pereira, "Stan: Exploiting shared interests without disclosing them in gossip-based publish/subscribe," in *Proceedings of the 9th International Conference on Peer-to-peer Systems*, ser. IPTPS'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863145.1863154

[10] S. Margariti and V. Dimakopoulos, "A study on the redundancy of flooding in unstructured p2p networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 28, pp. 214–229, 06 2013.

[11] J. A. Patel, E. Rivière, I. Gupta, and A.-M. Kermarrec, "Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems," *Comput. Netw.*, vol. 53, no. 13, pp. 2304–2320, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2009.03.018

[12] F. Rahimian, S. Girdzijauskas, A. H. Payberah, and S. Haridi, "Vitis: A gossip-based hybrid overlay for internet-scale publish/subscribe enabling rendezvous routing in unstructured overlay networks," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 746–757. [Online]. Available: https://doi.org/10.1109/IPDPS.2011.75

[13] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, ser. NGC '01. London, UK, UK: Springer-Verlag, 2001, pp. 14–29. [Online]. Available: http://dl.acm.org/citation.cfm?id=648089.747491

[14] V. Setty, M. van Steen, R. Vitenberg, and S. Voulgaris, "Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 271–291. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442626.2442644

[15] B. Wong and S. Guha, "Quasar: A probabilistic publish-subscribe system for social networks," in *Proceedings of the 7th International Conference on Peer-to-peer Systems*, ser. IPTPS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855641.1855643

[16] S. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," 01 2001, pp. 11–20.

[17] T. M. F. CIC. (2019, 06) Matrix specification 1.0. [Online]. Available: https://matrix.org/docs/spec

[18] I. Corporation. (2020) Distributed publish subscribe for iot. [Online]. Available: https://intel.github.io/dps-for-iot/index.html

[19] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, and Y. Psaras, "Gossipsub: Attack-resilient message propagation in the filecoin and eth2.0 networks," Tech. Rep., 2020.

[20] G. P. Jesi, A. Montresor, and M. van Steen, "Secure peer sampling," *Comput. Netw.*, vol. 54, no. 12, pp. 2086–2098, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2010.03.020

[21] B. Cohen, "Incentives build robustness in bittorrent," *Workshop on Economics of PeertoPeer systems*, vol. 6, 06 2003.

[22] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," jul 2020. [Online]. Available: https://github.com/ethereum/devp2p/blob/master/caps/eth.md

[23] M. Resende and P. Pardalos, *Handbook of Optimization in Telecommunications*, 01 2008.

[24] G. Urdaneta, G. Pierre, and M. van Steen, "A survey of dht security techniques," *ACM Comput. Surv.*, vol. 43, p. 8, 01 2011.

[25] A. Allavena, A. Demers, and J. E. Hopcroft, "Correctness of a gossip based membership protocol," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '05. New York, NY, USA: ACM, 2005, pp. 292–301. [Online]. Available: http://doi.acm.org/10.1145/1073814.1073871

[26] S. Voulgaris, D. Gavidia, and M. V. Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, p. 2005, 2005.

[27] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/1015467.1015471

[28] G. P. Jesi, D. Hales, and M. van Steen, "Identifying malicious peers before it's too late: A decentralized secure peer sampling service," in *First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, July 2007, pp. 237–246.

[29] A. Bakker and M. v. Steen, "Puppetcast: A secure peer sampling protocol," in *2008 European Conference on Computer Network Defense*, Dec 2008, pp. 3–10.

[30] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer, "Brahms: Byzantine resilient random membership sampling," in *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, ser. PODC '08. New York, NY, USA: ACM, 2008, pp. 145–154. [Online]. Available: http://doi.acm.org/10.1145/1400751.1400772

[31] H. Johansen, R. Van Renesse, Y. Vigfusson, and D. Johansen, “Fireflies: A secure and scalable membership and gossip service,” *ACM Transactions on Computer Systems*, vol. 33, 05 2015.
[32] A. Loewenstern and A. Norberg. (2008, jan) Dht protocol. [Online]. Available: http://bittorrent.org/beps/bep_0005.html
[33] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, “Eclipse attacks on bitcoin’s peer-to-peer network,” in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC’15. USA: USENIX Association, 2015, p. 129–144.
[34] T. Neudecker, P. Andelfinger, and H. Hartenstein, “A simulation model for analysis of attacks on the bitcoin peer-to-peer network,” in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015, pp. 1327–1332.
[35] C. Decker and R. Wattenhofer, “Information propagation in the bitcoin network,” in *IEEE P2P 2013 Proceedings*, 2013, pp. 1–10.
[36] S. Henningsen, D. Teunis, M. Florian, and B. Scheuermann, “Eclipsing ethereum peers with false friends,” in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2019, pp. 300–309.
[37] E. Foundation. (2020, Jul.) Ethereum wire protocol (eth). [Online]. Available: https://github.com/ethereum/devp2p/blob/master/caps/eth.md
[38] P. Silva, D. Vavricka, J. Barreto, and M. Matos, “Impact of geo-distribution and mining pools on blockchains: A study of ethereum,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 245–252.
[39] Streamr. (2017) Unstoppable data for unstoppable apps: Datacoin by streamr. [Online]. Available: https://streamr.network/whitepaper
[40] Digitransit. (2020) digitransit.fi. [Online]. Available: https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/
[41] Hope. (2020) The hope project. [Online]. Available: https://ilmanlaatu.eu/briefly-in-english
[42] S. Cohen, W. Nutt, and Y. Sagic, “Deciding equivalances among conjunctive aggregate queries,” *J. ACM*, vol. 54, no. 2, Apr. 2007. [Online]. Available: http://doi.acm.org/10.1145/1219092.1219093
[43] J. Leitao, J. Pereira, and L. Rodrigues, “Epidemic broadcast trees,” in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, ser. SRDS ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 301–310. [Online]. Available: http://dl.acm.org/citation.cfm?id=1308172.1308243
[44] N. Wormald, *Models of random regular graphs*, ser. London Mathematical Society Lecture Note Series. Cambridge University Press, 1999, vol. 267, pp. 239–298.
[45] Theory.org. (2019) Bittorrent protocol specification v1.0. [Online]. Available: https://wiki.theory.org/index.php/BitTorrentSpecification
[46] A. STEGER and N. C. WORMALD, “Generating random regular graphs quickly,” *Combinatorics, Probability and Computing*, vol. 8, no. 4, p. 377–396, 1999.
[47] F. B. Holt, V. Bourassa, A. M. Bosnjakovic, and J. Popovic, “Swan: Highly reliable and efficient network of true peers,” in *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*. Auerbach Publications, 2005, pp. 804–829.
[48] C. Cooper, M. Dyer, C. Greenhill, and A. Handley, “The flip markov chain for connected regular graphs,” *Discrete Applied Mathematics*, vol. 254, pp. 56 – 79, 2019.
[49] M. Castro, M. B. Jones, A. . Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman, “An evaluation of scalable application-level multicast built using peer-to-peer overlays,” in *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, vol. 2, 2003, pp. 1510–1520 vol.2.
[50] Y.-h. Chu, S. G. Rao, and H. Zhang, “A case for end system multicast (keynote address),” in *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’00. New York, NY, USA: Association for Computing Machinery, 2000, p. 1–12. [Online]. Available: https://doi.org/10.1145/339331.339337

PETRI SAVOLAINEN was born in 1979 in Nilsiä, Finland. He received an M.Sc. in computer science from the University of Helsinki in 2004, and is currently pursuing a PhD degree at the same university. Since early 2000, he has been active in various Finnish startups, assuming partner positions in companies that include Malibu Telecom Oy, Enporia Oy, Milvia Consulting Oy, and Spaceify Oy. As a researcher he has been involved with peer-to-peer networks research since 2008, and has been advising the Streamr project since 2018.

SANTERI JUSLENIUS was born in 1995 in Järvenpää, Finland. He received a B.Sc. in computer science from the University of Helsinki, and is currently a Master’s student at the same university.

He has been working as a Developer at TX in Helsinki since 2019. He initially worked in the Streamr Labs team, and joined the Streamr Network core team towards the end of 2019. He is driven to learn and understand technology on a deeper level, and its applications in decentralized edge and fog computing.

ERIC ANDREWS was born in 1988 in California, USA. He received an M.Sc. in computer science from the University of Helsinki in 2015, having specialized in algorithms, data analytics and machine learning.

He worked at Alma Media’s digital news operations for half a decade. In 2015, he joined Data in Chains in Helsinki, a company that later became part of TX. He has been working on the Streamr project since the very beginning, and is currently the Lead Developer for the Streamr project. He is interested in software architecture, machine learning, algorithms, and clean code.

MIROSLAV POKROVSKII was born in St. Petersburg, Russia, in 1985. He graduated with an M.Sc. degree in computer science at The National Research University ITMO in St. Petersburg in 2008.

He is a full-stack developer who has handled different roles for more than 14 years. His work experience includes Software Architect at bn.ru and Lead Sofware Developer at vdolevke.ru, as well as Full Stack Developer at Sujuwa (now TX) in 2015. Since 2018, he has been working on the Streamr project as part of the network team.

SASU TARKOMA (SMIEEE'12) was born in Helsinki, Finland in 1976. He is a Professor of Computer Science at the University of Helsinki, and Head of the Department of Computer Science at the same university. He has authored four textbooks and has published over 200 scientific articles. His research interests are internet technology, distributed systems, data analytics, and mobile and ubiquitous computing. He is Fellow of IET and EAI. He has nine granted US Patents. His research has received several Best Paper awards and mentions, for example at IEEE PerCom, IEEE ICDCS, ACM CCR, and ACM OSR.

HENRI PIHKALA was born in Helsinki, Finland in 1984. He holds an M.Sc. in computer science from Aalto University, graduating with honors in 2010. A software engineer turned serial entrepreneur, he started his first company in 2007 as a freelancer, while working as Lead Software Engineer in e-commerce and finance. In 2011, he co-founded Unifina and served as CTO in this algorithmic trading venture. In 2014, he co-founded Data in Chains and served as CTO, and in 2016 he was appointed CEO of the company. In 2017, he co-founded Streamr Network AG in Zug, Switzerland, and as its CEO, he crowdfunded and launched the Streamr open source project, and created the cryptocurrency Streamr DATAcoin. In 2019, he started TX, a company focused on blockchain enterprise services.