Master's thesis

Master's Programme in Computer Science

WebSocket vs WebRTC in the stream overlays of the Streamr Network

Santeri Juslenius

May 26, 2021

Faculty of Science

University of Helsinki

Supervisor(s)

Dr. S. Varjonen, MSc. Eric Andrews

Examiner(s)

Prof. S. Tarkoma

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Santeri Juslenius

Työn nimi — Arbetets titel — Title

WebSocket vs WebRTC in the stream overlays of the Streamr Network

Ohjaajat — Handledare — Supervisors

Dr. S. Varjonen, MSc. Eric Andrews

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 26, 2021	90 pages

Tiivistelmä — Referat — Abstract

The Streamr Network is a decentralized publish-subscribe system. This thesis experimentally compares WebSocket and WebRTC as transport protocols in the system's d-regular random graph type unstructured stream overlays. The thesis explores common designs for publish-subscribe and decentralized P2P systems. Underlying network protocols including NAT traversal are explored to understand how the WebSocket and WebRTC protocols function. The requirements set for the Streamr Network and how its design and implementations fulfill them are discussed. The design and implementations are validated with the use simulations, emulations and AWS deployed real-world experiments. The performance metrics measured from the real-world experiments are compared to related work.

As the implementations using the two protocols are separate incompatible versions, the differences between them was taken into account during analysis of the experiments. Although the WebSocket versions overlay construction is known to be inefficient and vulnerable to churn, it is found to be unintentionally topology aware. This caused the WebSocket stream overlays to perform better in terms of latency. The WebRTC stream overlays were found to be more predictable and more optimized for small payloads as estimates for message propagation delays had a MEPA of 1.24% compared to WebSocket's 3.98%. Moreover, the WebRTC version enables P2P connections between hosts behind NATs. As the WebRTC version's overlay construction is more accurate, reliable, scalable, and churn tolerant, it can be used to create intentionally topology aware stream overlays to fully take over the results of the WebSocket implementation.

ACM Computing Classification System (CCS)

Networks → Network types → Overlay and other logical network structures → Peer-to-peer networks

Networks → Network protocols → Application layer protocols → Peer-to-peer protocols

Networks → Network protocols → Transport protocols

Avainsanat — Nyckelord — Keywords

Overlay Networks, publish-subscribe, WebRTC, WebSocket

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking study track

Contents

1 Introduction

Publish-subscribe systems have been used as engines for messaging in distributed systems for many years. They provide good inherent scalability and allow the subscribers and publishers of messages to be decoupled. These properties make the publish-subscribe pattern an outstanding candidate for networking in the Internet of Things (IoT). Publish-subscribe systems are often built on top of message brokers. Modern Message brokers are often formed of distributed components [35] used to validate, transform and route messages in and out of systems. Currently centralised, cloud based message brokers are the most commonly used publish-subscribe systems for IoT. However, these types of solutions are known to struggle as the numbers of subscribers and data volumes increase. In recent years, IoT as an industry has grown rapidly with no end in sight. With some estimates for the amount of IoT devices nearing 75 billion by 2025 [8], new more scalable, available and robust publish-subscribe systems are needed.

1.1 Centralised vs decentralised

There are many pitfalls for centralised cloud solutions as a whole. The most obvious ones are vendor lock, pricing fluctuations and single points of failure. Modern clouds providers offer many solutions that improve the reliability and robustness of their systems. However, single points of failures that may shut down large parts of a publish-subscribe system still remain. In many IoT solutions this can be unacceptable. In a scenario where a sensor sends a message that requires immediate action, in the worst case a failing centralised broker could lead to a disaster. Most of such technical failures would generally only lead to economic damages. However, as IoT expands to new domains such as the healthcare sector, lives could be at risk.

During the early 2000s the publish-subscribe and peer-to-peer (P2P) research roared concurrently. This lead to many of the likely issues with centralized message brokers being recognized early on and various decentralised P2P publish-subscribe systems were proposed [6, 17, 71, 95, 14, 61]. More recently, the high availability and scalability of such systems has started to see interest in the IoT field. Decentralized publish-subscribe systems are often able to keep the amount of connections per broker constant regardless of

the amount of subscribers in the network. This feature is very attractive for any use case that requires high scalability. More specific examples of such systems are blockchains. Blockchains such as Ethereum [12] have been using P2P overlays for message propagation in their network.

Another new occurrence of decentralized systems in practice are P2P social media platforms. Such platforms have started popping up as competition to the centralised giants [50]. As more P2P based solutions are starting to appear in the market there will be an increasing need for reliable P2P based publish-subscribe systems to support new applications. This has already caused a comeback in the decentralized publish-subscribe system research. The search for the best practical implementations for different use cases has already begun. For example, there was a race to find a replacement for Ethereum's Raiden network during the development of Ethereum 2.0 which was won by GossipSub [86].

Decentralized publish-subscribe systems come with their own downsides. The latency in P2P overlays is often worse on average and more unpredictable than in centralised solutions. This is often caused by messages being propagated via multiple nodes in the network before reaching their final destination. Nodes can also join and leave the network without warning (churn) causing temporary instability in message propagation. This ends up making the previously described IoT warning system an unlikely candidate to use public decentralised public-subscribe systems. Churn can also be impossible for low computing and battery powered IoT devices to handle, as the churn related control traffic can account for a major part of the network traffic. Moreover, decentralized systems tend to be vulnerable to various forms of attacks [5]. In the worst cases, these attacks can cut off significant parts of the networks, bring entire networks down or take control of them. This makes attack resilience one of the most important qualities of any decentralized system.

1.2 The Streamr Network

The Streamr Network [70] is a decentralized publish-subscribe network. The network's design addresses the previously mentioned challenges of decentralized networks. The goal of the Network is to enable a decentralized marketplace for real-time data by using the Ethereum blockchain [12] for bootstrapping, validation and transactions. The Streamr Network uses topics for publishing and subscribing to messages similarly to systems such as Apache Kafka [27]. Each topic referred to as stream is a separate overlay meaning that nodes can never receive content traffic that they are not interested in. On top of

avoiding uninteresting content traffic, the nodes do not receive any uninteresting control traffic due to the use of trackers. However, each node is tasked with contributing some of their uplink bandwidth to forward any messages that they receive which increases the system's bandwidth consumption. This is key for the network to scale without limit.

This thesis explores the requirements, design and experimental validation of the Streamr Network as presented in its Corea release whitepaper [70] and expands on the research by comparing the design and performance of the Corea Release's WebSocket based stream overlays to the upcoming Brubeck milestone's WebRTC based stream overlays. The Corea release of the Streamr Network was only meant for internal data center setups and did not allow users to run nodes for decentralization. The network's overlay connections were formed using the WebSocket protocol and forming P2P connections between nodes behind NATs was not possible. The upcoming Brubeck release will take the first steps toward decentralization by making it possible for users to run broker nodes. To achieve this, users must be able to run the nodes behind NATs. A preliminary choice was made to enable P2P NAT traversal by replacing the WebSocket protocol with WebRTC and the decision is validated with the experimental results of this thesis. The use of WebRTC comes with the additional benefit of making it possible to run broker nodes inside browsers without significant development efforts to improve the scalability of the network.

In chapter 2, the background of the Streamr Network will be explored. This includes exploration of publish-subscribe systems, decentralized P2P networks and the WebSocket and WebRTC protocols including the underlying protocols that form them. Chapter 3 will go over the requirements for the Streamr Network and explore how these requirements have been addressed in related work. In chapter 4, the design of Streamr Network is discussed. On top of this, the differences between the WebSocket and WebRTC stream overlay implementations are described. In chapter 5, experiments used to validate the design of the Streamr Network are presented. This includes simulation experiments to prove that the graphs generated by the trackers in the Streamr Network are d-regular random graphs. Emulation experiments were performed to measure the bandwidth efficiency of the Streamr Network are presented. Emulation experiments were also used to measure the performance of the WebRTC and WebSocket stream overlays under packet loss. Next, AWS based Real world experiments were performed to measure the scalability and performance of the two implementations of the Streamr Network. Finally, the results of the experiments are compared to those of related work. Chapter 6 recaps the findings of the research and the thesis is concluded in chapter 7.

2 Background

Building decentralized overlays is not easy. Many things need to be considered from the underlay network up and the justifications for key decisions need to be understood. Whenever a team arrives at a crossroads between technologies it is important to know of the presence of any trade-offs. Understanding the used technologies will save countless man-hours and could save a project from failing.

In this chapter the main underlying technologies for the Streamr Network will be explored. This will include the details for the underlay and overlay networking technologies and methods used by The Streamr Network and related work. On top of this, the basics of the Ethereum blockchain and its smart contracts' impact on decentralization will be explored.

2.1 Publish-subscribe model

The publish-subscribe model has become a staple for distributed systems. On top of providing great scalability, publish-subscribe systems tend to expose easily programmable APIs and configurable guarantees for message delivery. The ease of use, once setup, of publish-subscribe systems is no doubt one of the main factors for their success.

Individual publishers and subscribers are decoupled publish-subscribe systems. Meaning that they do not necessarily know of each other's presence. Publishers only publish events to the systems while the subscribers receive the events they are interested in. The publish-subscribe system itself is responsible for delivering the published events to their corresponding subscribers. Decoupling increases the scalability of the system as there are no dependencies between the parties in the system. [24]. It also makes the system very asynchronous in nature making it excellent for cases such as IoT.

The decoupling of the users does have its downsides. It makes it difficult for publishers to know if their messages ever reach subscribers. There are ways to alleviate this and systems such as Apache Kafka [27] provide guarantees for message delivery. However, implementing the guarantees increases the overhead of the system. For P2P overlay-based solutions sending acknowledgements and resend requests can lead to the control traffic taking the majority of the systems bandwidth. Publish-subscribe systems also tend to face

problems as the number of publishers and subscribers increase. It is difficult for brokers to maintain connections and to propagate all of published messages. Decentralized solutions can reduce the amount of required connections for brokers by requiring publishers and subscribers to propagate messages to a fixed number of nodes. However, this solution may cause further problems with another known issue with publish-subscribe systems. Centralized publish-subscribe systems run into problems with load surges. If events need to be forwarded to thousands of subscribers using unicast, the linear bandwidth requirements of the system will become its downfall. If all publishers and subscribers contribute uplink bandwidth by forwarding all received messages the bandwidth consumption of the system will be constant. However, in such a solution the maximum throughput of individual subscribers and publishers will be more limited. Overall, as in any distributed system, publish-subscribe systems come with tradeoffs in system properties.

2.1.1 Traffic confinement

When building decentralized publish-subscribe systems, it is important limit the amount of traffic that traverses through each peer. It is especially important that peers do not receive too much traffic that they are not interested in themselves. Limiting the traffic to only those peers that are interested in it is called traffic confinement. In the most simple and inefficient implementation of a decentralized publish-subscribe system presented in [6] the peers would form a mesh (random graph) topology. Each peer would propagate events to all of its connections and would select from those events only the ones that it is interested in for further processing. Such a system is obviously very wasteful in terms of overall resource use. This makes traffic confinement one of the most important design points in terms of overall performance of the system. The measurement of the level of success in traffic confinement in a decentralized system is defined as noisiness in [57]. Any traffic that a peer is not interested is defined as noise.

Achieving noiselessness is important for decentralized systems. However, there are differences in difficulty between system types on achieving it. It is easier to implement noiselessness in decentralized topic-based publish-subscribe systems than in content-based ones. Topic based systems define the traffic's route much more implicitly than content-based systems. The causes for this will be explored in the next subsection. Luckily there are strategies on how to implement traffic confinement in decentralised publish-subscribe systems. These strategies are defined in [6].

Interest Clustering Strategy is defined as the first step. Subscribers should be arranged in a way where peers with common interests are clustered. Once a single peer in the cluster receives an event it should be able to forward the message to all peers in the cluster. Optimally, the forwarding should be limited to the peer in the cluster. Thus, the goal of the strategy is to have all the subscribers interested in an event in the same cluster. By doing this the reliability of the system is improved.

Once an event has reached one member in the interest cluster, there should be a common strategy on how to forward the event to all peers. This strategy is called the Inner-Cluster Dissemination. There are no strict guidelines on how to implement this strategy in [6]. However, it is important that such a strategy can exist in the system. The strategy in its simplest form could be flooding based, where all peers propagate all received messages to their neighbors.

The last strategy is called Outer-Cluster Routing. It defines how an event is routed to its interest cluster in the system. The goal is to contact as few peers as possible to achieve this. The goal to achieve full noiselessness is to contact zero uninterested peers in the system. This strategy is challenging for content-based systems to achieve. On the other hand, Topic-based systems can achieve this quality among the other two quite inherently. The reasoning for this will be seen in the next subsection.

2.1.2 System Types

As publish-subscribe systems are distributed systems in nature, many different approaches to the model have been proposed. Most of currently used designs are centralized and focus on cloud computing. However, over the years many decentralized solutions have been proposed. These solutions will be explored in more detail in chapter 3. This section will focus on the different publish-subscribe system types that exist regardless of the system being centralized or decentralized. The main difference in these system types is how the subscribers specify their interest to data.

In general, publish-subscribe systems are split into two categories: Content and Topic based systems. In Content based systems subscribers let the system know that they are interested in certain types of events based on filters. Publishers send messages to the system without labeling the data to any specific category. When the system receives the published messages, it applies the subscribers' filters to the data. The messages are then passed to the interested subscribers based on the filters. This makes the type excellent

for warning systems and there are many different solutions deployed in production use. Some decentralized content based publish-subscribe systems have been proposed [28] [93]. Decentralized content publish-subscribe networks have not seen much real-world use. The nature of the content type system causes each message to be propagated to many brokers that are not interested in the message as there are no interested subscribers connected to it. This feature of the system makes it quite heavy for real. For example, in Meghdoot [28] in the worst case 15% of the nodes need to be contacted to deliver a message to a subscriber. Ferry [93] was designed as a new decentralized content publish-subscribe network to improve many of the performance flaws in Meghdoot's message propagation. However, even in Ferry 10% of nodes need to be contacted to deliver a message in the worst case. This noisiness due to the Outer-Cluster Routing in the network makes the system model quite poor for high throughput use.

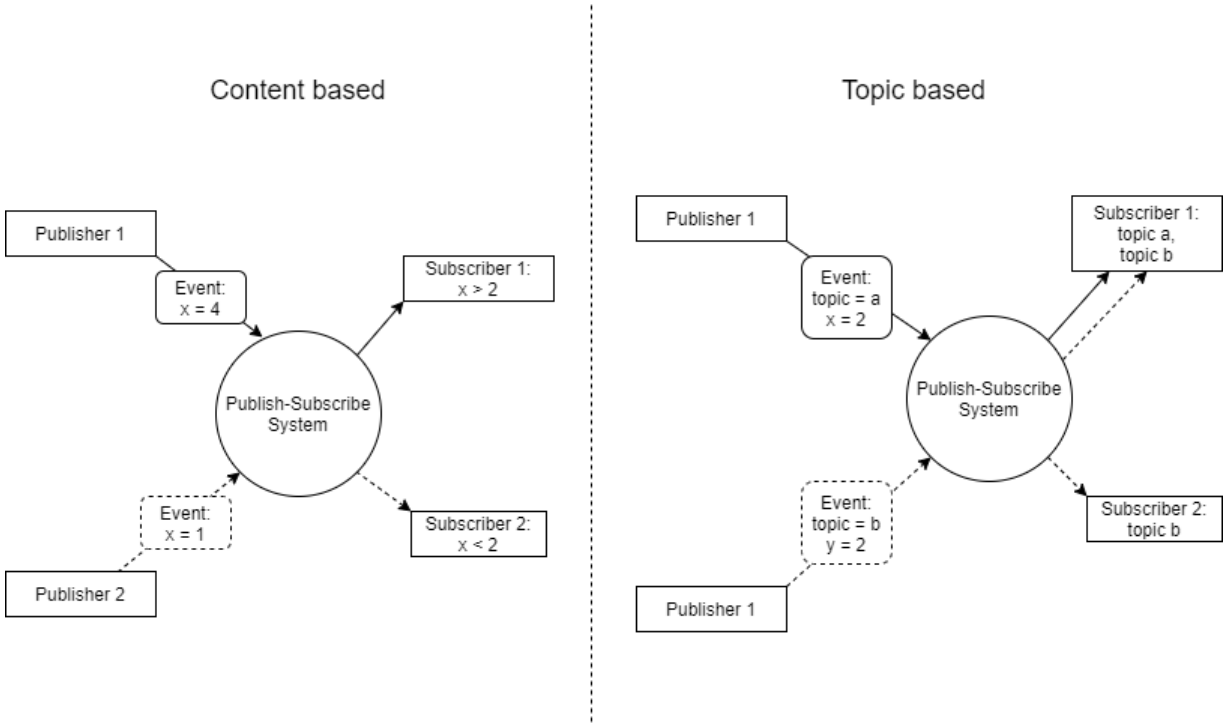

Figure 2.1: Content and Topic Based Publish Subscribe

In topic-based systems subscribers receive messages based on keyword defined topics. To achieve this all publishers have to set the key of the topic to any events that they send to the system. This quality makes topic-based publish-subscribe systems excellent for real-time data streaming. Similarly to content-based systems, topic-based systems have seen wide adoption. The basic differences between content and topic-based systems can be seen in Figure 2.1. A notable example of a widely deployed centralized topic-based

publish-subscribe system is Apache Kafka. Importantly for this thesis, Streamr Network is a topic-based publish-subscribe system. Being topic-based allows nodes to only receive and propagate messages which they are interested in, making the system noiseless. As the subscribers in the system have clear and defined interests it is easy to implement all the traffic confinement strategies. The design of Streamr Network will be discussed in detail in chapter 4.

Alongside the more popular topic and content-based system there exists a hybrid solution between the two. This system type was originally called type-based [24] publish-subscribe, but it has later been coined as hybrid. In these solutions messages are published into topics but the subscriptions to the topics are content-based. This can be particularly useful as subscribers may not be interested in receiving all the events published to a topic.

2.2 Underlying Networking Technologies

Overlay networks are networked systems by definition. When such systems are being built and designed, thought must be given into which network protocols should be used. Network protocols differ in their performance, reliability, and ease of use. This means the used protocols often change depending on the use case.

Client-server applications have reached the point where it is easy for any developer to build applications. Prepackaged HTTP client and server solutions remove the need to take the underlay into account. Application layer networking protocols for P2P solutions have also been introduced. These solutions have made it possible for anyone to program their own P2P video calls and chat rooms. However, centralized rendez-vous points are required for these solutions to function in the presence of NATs. Devices running nodes in P2P overlays tend to also have very heterogeneous properties. Some nodes have better down or up link capacity and some have more memory and CPU power than others. Some networking protocols especially in the transport layer can be used to alleviate these challenges.

In this section, the WebSocket and WebRTC application layer protocols used by The Streamr Network will be explored. To understand any measured performance differences between the solutions, it is necessary to go through the differences in the used underlying protocols. The challenges of Firewalls and NAT traversal will also be addressed.

2.2.1 TCP

Transmission Control Protocol (TCP) [83] is the most commonly used transport layer protocol in the TCP/IP stack. TCP provides reliable, in-order and bidirectional data transportation between two computers as byte streams. There are several mechanisms added to TCP that add flow and congestion control to the protocol. The different congestion control methods are especially important, as the internet in its current form would collapse without them.

A TCP connection is formed by using a three-way handshake as seen in Figure 2.2. First, computer A sends a SYN-packet to computer B. Once computer B receives the SYN-packet it sends a SYN-ACK packet to computer A letting it know that the SYN packet was received. Once A receives the SYN-ACK, it sends an ACK-packet back to B. Once the ACK is sent the connection is established from A's perspective and it may start sending bytes. The connection is established from B's perspective once the ACK is received. As the connection is being formed the nodes may also negotiate the maximum segment size (MSS). The MSS is used to define the maximum amount of data that a single TCP segment can contain. If MSS negotiation is used computer A will send the MTU of its outgoing link with the SYN packet. B will send a response with the SYN-ACK packet containing the corresponding information. Once the connection is established in either end, the endpoints compare the received and their own outgoing links MTU values and select the minimum as the MSS.

TCP connections are closed using a four-way handshake. The process is started by A sending a FIN-packet to B. Once the packet is received, B sends an ACK-packet and a FIN-packet to A. After A has received both packets it sends an ACK-packet back to b and starts a timer before closing the connection. The timer is set to make sure that A receives any remaining data in flight by B. B closes the connection once it receives the ACK sent by A. The connection termination process consists of two FIN-ACK sequences initiated once by both ends of the connection.

During the data transfer period in TCP many mechanisms are used to ensure reliability. Each packet sent through a TCP connection is sequenced. This makes it possible to ensure that the data is received in-order. Each packet is acknowledged back to the sender by sending an ACK-packet containing the sequence number of the next packet that receiver is expecting. If the sender does not receive an ACK for a packet within a timeout period it will resend the packet to the receiver. The TCP packets also contain checksums that

can be used to detect faults in the received packets. TCP also implements a push function that can be used to push the buffers to the next layer immediately instead of waiting for the buffer to fill up. There are many implemented modifications to the data flows in TCP. These modifications improve the performance of TCP and avoid congesting the network.

Congestion control is an area of TCP that has seen many improvements over the years. Some of the original tools for congestion control were congestion windows and slow starts. Congestion Windows (cwnd) and Receiver Windows (rwnd) [1] determine how much data can be in-flight in a TCP connection. The minimum of the two is used as the governing value. Slow Starts [1] are used whenever a connection is initiated, a Retransmission Timeout (RTO) runs out or an open connection has not been used for a long time. The main purpose of Slow Starts is to determine the network capacity. To achieve this, whenever a sender receives an ACK it will increase the size of the cwnd by one MSS until a lost segment is detected. With this strategy the size of the MSS is approximately doubled per round time trip (RTT). Once loss is detected in most implementations' half of the cwnd is saved to the ssthresh value.

In TCP Tahoe [72] an early congestion control algorithm, uses duplicate ACKs for loss detection. Whenever a receiver detects an out of sequence segment, it sends a duplicate ACK with the sequence number of the message that it is expecting to receive in sequence. In Tahoe, once three duplicate ACKs are received by the sender a Slow Start is initiated. The cwnd is initially lowered to the initial MSS and a Slow Start is initiated until the cwnd reaches the ssthresh value. Finally, TCP enters congestion avoidance where the cwnd grows linearly until the next lost segment.

TCP Tahoe's approach to congestion control tends to lead to bottlenecks. If devices are connected to networks with bad reception lost segments are bound to be more common. This leads to Slow Starts being initiated more often, limiting the throughput of the protocol. The issue was already recognised even as Tahoe was released and TCP Reno [56] was introduced as an alternative. TCP Reno employs the first implementation of fast recovery. Upon receiving 3 duplicate ACKs, instead of setting the cwnd to its initial value, the cwnd value is halved and congestion avoidance is started immediately.

Over the years TCP has seen many further improvements to its congestion control. Many of them use duplicate detection as their loss detection mechanism. One of the problems with this approach is that duplicate ACKs are only able to communicate one segment of lost data at a time. This results in throughput issues if there are many dropped segments as the endpoints need to wait for an RTT trip for each resend. Many modern fast recovery

techniques alleviate this problem by sending multiple resend segments at a time. However, many resends per RTT may end up being redundant if only gap fills are required. Selective Acknowledgements (SACK) [51] were designed to combat these issues. SACKs make it possible for the receiver to acknowledge many lost segments with a single ACK. When duplicate ACKs are sent with SACK, up to 3 segment ranges can be marked as received. This makes it possible for the sender to only resend the segments that were dropped. Therefore, the use of SACK leads to significant increases in the performance of TCP as throughput bottlenecks are reduced.

2.2.2 SCTP

The Stream Control Transmission Protocol (SCTP) [59] is a transport layer protocol, originally designed for Public Switched Telephone Network (PSTN) signaling over IP. The main motivation for SCTP is to address some of the limitations of TCP in PSTN and in general. TCP limits all delivery to be in-order. For many applications only reliable delivery or partial order of messages is required. For such applications TCP ends up being a performance bottleneck. The byte stream orientation of TCP also requires developers to distinguish between separate messages in the byte stream. To ensure that messages are completed in a reasonable time, the use of the push function in TCP is required. To work around these two challenges, developers have often ended up building their own custom UDP solutions.

One of the key differences between SCTP and TCP is that SCTP sends streams of messages instead of bytes. This solves the issues related to TCPs byte streams. Despite this key difference, sending acknowledgements back to the sender is done in a similarly to TCP as SCTP uses SACKs by default. SCTP adds configurability to the ordering of messages and reliability of the connection. Contrary to TCP, it is possible to send unordered and ordered streams reliably in SCTP. This is done by separating the Transmission Sequence Number (TSN) from any Stream Sequence Numbers (SSN). By doing this any received messages can be acknowledged back to the sender based on the TSN, leaving the ordering to be done based on the SSN.

TCPs three-way handshake is also vulnerable to SYN flood attacks [23]. These attacks often lead to server machines leaving insecure half open connections on a single port. If the server is unprotected, the attack most commonly leads to the server being unable to form new connections or to a server crash. SCTP uses a four-way handshake to prevent

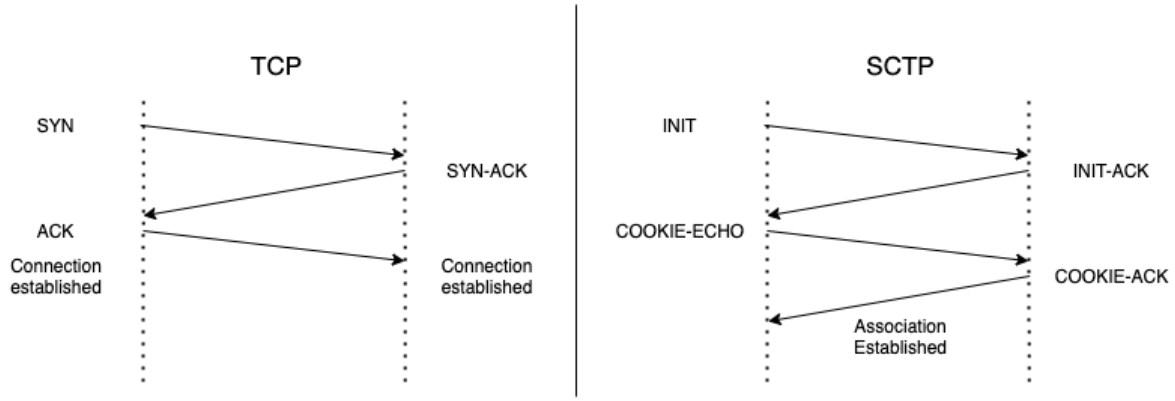

Figure 2.2: TCP's Three Way Handshake vs SCTP's Four Way Handshake

SYN flood attacks as seen in Figure 2.2. Outside of this improvement, SCTP does not have any significant security benefits over TCP.

Until recently SCTP has beat TCP in terms of reliability in one significant way, SCTP allows the use of redundant paths for message delivery with multihoming by default. When multihoming is used, SCTP endpoints bind connections to multiple local IP addresses. This makes it possible for messages to take multiple different paths in the underlay during transportation. With this approach if a message is lost, for example in a router along its path, the message in the other path may still reach its destination. However, in 2020 the first official multipath TCP standardization was released [26]. This equalizes the playing field in terms of path redundancy.

The congestion control used in SCTP is based on TCP mechanisms [82]. Almost any congestion control mechanism that TCP implements can be used with SCTP. However, SCTP does have some differences to TCP. As mentioned before, the use of SACKs is mandatory in SCTP. This makes SCTP a more robust protocol when many gap fills are required and avoids entering slow starts. During congestion avoidance SCTP does not allow its cwnd to be increased before it is being fully utilized. This mechanism is great for avoiding network congestion. The last notable difference is that SCTP enters fast recovery after four duplicate ACK compared to TCPs three. Despite this, SCTP begins to retransmit data based on SACKs after three duplicate ACKs have been received.

The benefits of SCTPs congestion control and retransmission methods are notable in the experiments run in [82]. The authors showed that under normal network conditions TCP outperforms SCTP by approximately 10% in transmission times. However, when 5% packet loss was introduced into the network, SCTP outperformed TCP by 5 times in the same metric. The authors also compared the throughput of SCTP and TCP in HTTP file

transport in normal and impaired network conditions. The sizes of the files varied from 100 bytes to 100 kilobytes. The experiments included the use of multistreaming in SCTP with 1, 10 and 100 streams used. In normal network conditions, TCP outperformed the single stream SCTP quite significantly as the size of the files grew. However, 10 stream STCP's performance was almost equal to TCP with 100 stream SCTP significantly outperforming TCP. With impaired network conditions TCP outperformed single stream SCTP but as the file sizes grew, SCTP outperformed TCP by a large margin. 10 and 100 stream SCTP outperformed TCP with all file sizes. Quite interestingly, the single stream SCTPs performance reached the multistream variants at 100 kilobyte file sizes. Under impaired network conditions, SCTP's higher performance is explainable by its more efficient congestion control and avoidance. TCP's higher performance under normal network conditions can be explained by SCTP's higher overhead.

In [48] the authors performed experiments to compare the performance of single stream SCTP, TCP and UDP in a dumbbell topology with bottlenecks. The simulation experiments showed that similarly to [82], SCTP outperformed TCP in networks with bottlenecks. The throughput of SCTP was higher than that of TCP. Also, as data rates grew higher TCP's end-to-end latencies kept rising. SCTPs end-to-end latencies outperformed TCP by becoming stable once the data rates reached 2Mb/s. The measured variance in the end-to-end latencies (jitter) also becomes stable with higher data rates in SCTP. Lastly, SCTP significantly beat TCP in packet delivery rate (PDR). TCP's PDR became linearly lower as the data rates grew whereas SCTP's PDR once again remained stable. The one area where TCP beat SCTP by a larger margin was fairness. In the experiments, fairness was defined as the difference of packets received by to destinations. TCP's fairness was near 0 until high data rates where it started to jump from 1 megabytes to -1 megabytes. Once again, SCTP's fairness became stable after 2 Mb/s data rates, at the fairness value of 2 megabytes.

Overall, the results of [82] and [48] show that SCTP mostly outperforms TCP in restricted networks. However, TCP does seem to be more efficient in good and stable network conditions. Initially, these results indicate that for heterogeneous P2P networks, SCTP should be the better candidate. However, TCP is expected to outperform SCTP in centralized datacenter setups while SCTP should beat TCP in heterogeneous networks. However, SCTP has a problem with CPU load in low latency environments. The check sums used by SCTP can exhaust a single core's capacity completely. This has led to the creation of non-standard implementations of SCTP where the checksums are not used [33]. Despite

this the most recent SCTP standards still mandate the use of the check sums. The first attempt to standardize the check sums as optional in the message payloads was in 2002 [77] but the proposal never passed. While the CPU problem with SCTP is less prevalent with larger end-to-end latencies, it is throughput bottleneck of the protocol in low latency setups.

2.2.3 DTLS

Datagram Transport Layer Security (DTLS) [64] is a protocol that provides encryption for datagram based protocols in the transport layer. The most used datagram-based protocol is User Datagram Protocol (UDP). UDP datagrams are sent to a receivers address and port without any expectations of encryption, acknowledgements, or congestion control. These features make UDP unreliable, insecure and capable of taking down routers and networks. SCTP introduced reliability and congestion control to datagram-based protocols. However, SCTP did not address the privacy or security of the datagrams.

Transport Layer Security (TLS) [22] protocol is used as the basis of DTLS. TLS was originally designed and built to make client-server HTTP traffic more secure. However, TLS can be used in all TCP connections to provide end-to-end security. The goals of TLS are to provide privacy of connections, interoperability of applications, extensibility of the protocol and such efficiency that the performance costs of security are not detrimental to the quality of service of applications.

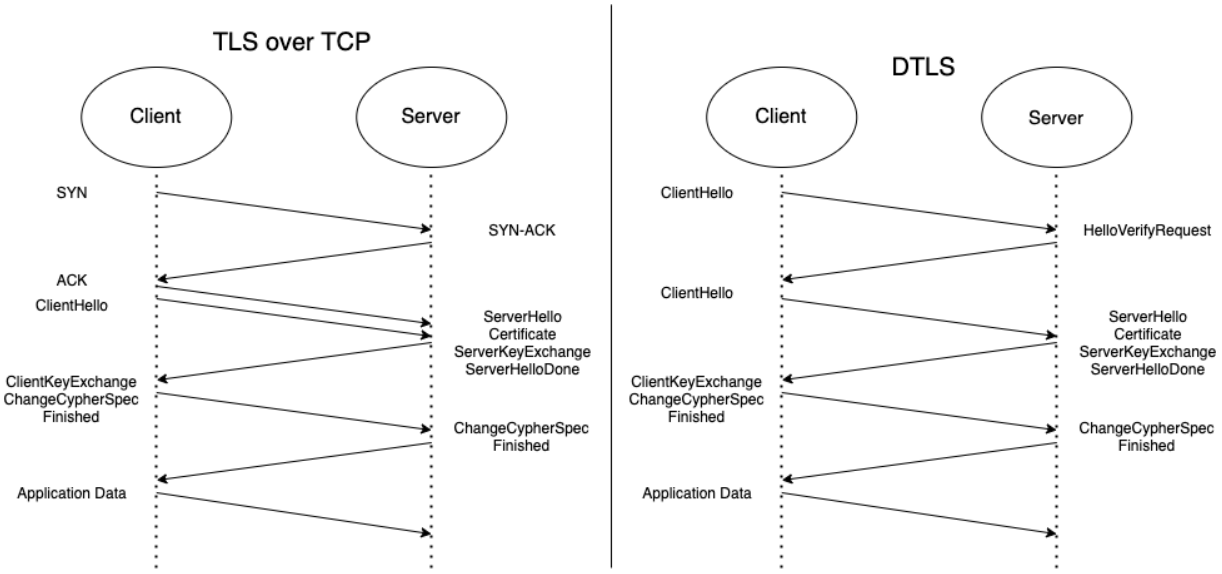

Figure 2.3: TLS over TCP and DTLS full handshake comparison

When TLS is used with TCP, a TLS handshake is started immediately after the connection is established in the client's perspective. No data should be passed through the connection until the TLS handshake is completed. As seen in Figure 2.3 the process is started by the client and server by exchanging Hello messages. The Hello messages are used to decide which algorithms to use, to exchange random values and to decide if the session should be continued. Next the peers exchange cryptographic parameters to generate a premaster secret. Once the premaster secret is generated, cryptographic information and certificates need to be exchanged for the peers to authenticate themselves. After authentication, a master secret is generated using the random value in the hello messages and the premaster secret. Finally, the security parameters are moved from the handshake layer to the record layer and the peers verify that the security parameters have been calculated correctly and that the handshake was completed without manipulation. TLS 1.2 [22] introduced a faster way to implement the TLS handshake by using a three-way handshake. This was further improved in TLS 1.3 with a two-way handshake [63]. As TLS 1.3 is still quite recent, an official DTLS 1.3 protocol has not come out yet but is being worked on.

DTLS has to account for reliability as UDP does not provide any such guarantees [64]. In TLS, all decryption is based on sequence numbers. If the sequences arrive in the wrong order the decryption will fail. Furthermore, if TLS handshake messages are lost the session will be stopped. The handshake messages also tend to be quite large. This may lead to fragmentation in the lower layers of ip. This is a problem as UDP cannot reassemble the packets on its own. DTLS has to include mechanisms that enable TLS without TCP by addressing the challenges mentioned above. On top of this, any cryptographic measures that are used to retain context between records in TLS cannot be used. This must be done as ordering cannot be guaranteed.

To address message reordering and anti-replay DTLS implements its own sequence numbering and duplicate detection [64]. Using the sequence numbers, if a message arrives out of sequence during handshaking it is queued for the future. Packet loss is accounted for by using retransmission timeouts. If acknowledgements for handshake messages are not received, the message will be retransmitted after a timeout runs out. DTLS also implements a stateless ClientHello as the first step of connection establishment as depicted in Figure 2.3. The server is expected to respond with a HelloVerifyRequest message to the initial ClientHello. This is done for protection against Denial of Service (DoS) attack.

DTLS can also be used with SCTP [84]. While it is possible to simply use TLS to encrypt SCTP messages, the use of TLS forces SCTP to send all messages reliably in order. This

makes DTLS the better candidate for use in SCTP as it supports TLS encryption without requiring inorder sequencing of messages [85]. However, whenever DTLS requires messages to arrive in order, for example during the initial handshake SCTP should be used to handle it.

There are two separate ways to use DTLS with SCTP, either DTLS over SCTP or SCTP over DTLS. If DTLS is used over SCTP, one DTLS connection cannot be used for multiple SCTP associations between two machines. Initially, an application passes the data to DTLS where it is encrypted. From there the data is passed on to SCTP which passes the data on to the lower layers to be transported. The receiving endpoint does the process in reverse to pass the data to its application layer [85]. If SCTP is used over DTLS, application data is first passed to DTLS. The main benefit of this implementation is that it becomes possible to use a single DTLS connection to support multiple SCTP associations between two machines [84].

2.2.4 WebSocket

The WebSocket Protocol [25] is an application layer protocol that uses underlying TCP connections. Traditionally applications that use HTTP for networking have only been able implement pull based models. This means that in a client-server model only the client can initiate any interactions. This comes with some problems for the server. For example, the server is forced to keep open multiple TCP connections. One connection for each incoming request and one for sending all outgoing responses. In an optimal solution one TCP connection would be used to handle all interactions between the server and client. WebSocket enables this by making it possible to send unsolicited messages between the client and server. On top of reducing the number of connections WebSocket significantly reduces the header overhead of HTTP reducing bandwidth use.

The WebSocket protocol only adds minimal functionality to TCP and acts only as thin layer above TCP. WebSocket adds web origin security, making it possible for servers to deny clients based on their address [7]. WebSocket also implements a naming policy which makes it possible to run multiple services on single ports and multiple host names on single addresses. All sent data is also packaged into WebSocket's own standardized frames. These frames include the control frames ping, pong and close along with the dataframes. The WebSocket protocol also requires additional handshake protocols for forming and closing the connections.

WebSocket uses HTTP for sending initial handshake messages. The client initiates the connection by sending a request to a server's address and port. On top the address information the client also has to specify the resource name it wishes to access along with its origin. A secure flag may also be set to request using TLS. There must always only be one connection to a single address and port. If a connection already exists or is being opened but the client wants to access a new resource, it must associate the new resource to the existing connection. This ensures that there is only one TCP connection between two machines at once. Once the client sends the initial handshake message it enters the connecting state and waits for a response from the server.

The server will process the handshake message upon receiving it parsing through the fields to see if all data is valid. If the data is valid the server must complete a series of steps before opening the connection. During this phase the server checks if it wants to accept the client's origin. Any subprotocols or extensions that are requested by the client must be validated. If the server does not support requested subprotocols or extensions, it returns a null value for the fields. The WebSocket version must be validated as well and if it is not supported the connection is aborted. The requested resource name is also validated. If the resource is not available, the handshake is aborted. If a secure connection is requested a TLS handshake is initiated after which the client is sent a response to the client, sets its connection state to open and may start sending data. Once the client receives the response it goes through the fields and validates them. If the client does not accept the received values, it must fail the connection. If all received values are valid the connection is set to the open state and the client may start sending data.

Although WebSockets do not add much on top of TCP, its handshake period takes four times as long [73]. One of the main reasons for this is that the TCP connection must be negotiated before the WebSocket handshakes can be started. However, once data is being sent through the connection the performance differences are much smaller and unnoticeable in practice. In overlay use, the increase in the time to form connections may cause problems in the presence of churn. Although, for stable connections there should be no performance degradation when compared to TCP.

2.2.5 NAT traversal

In P2P solutions each node has to act as a server and a client and servers are found in the Internet based on IP addresses. In the Internet, IPv4 addresses have ran out a

long time ago. IPv6 [30] was introduced to alleviate this issue by increasing the address space from 22^{32} to 2^{64}. However, the transition from IPv4 to IPv6 has still not happened completely. Network Address translation (NAT) [74] is used to alleviate the problem of the limited scope of IPv4 addresses. NATs allow the same IPv4 addresses to be reused. Basically, NATs connect private networks to public networks by translating addresses. NATs come with significant downsides. NATs are single points of failure that may end up disconnecting large parts of the internet from each other. Moreover, NAT traversal makes it much more difficult to find servers that are placed behind NATs based on their address in the internet. What's worse, NATs were eventually introduced in IPv6 due to mostly misguided security concerns. This means that NAT traversal will be needed even after IPv4 has completely faded away. Currently there are four typologies of NATs in use: Full Cone, Address Restricted Cone, Port Restricted Cone and Symmetric NATs [21].

Full Cone NATs map a private hosts address and port directly to a public NAT pairing. This means that any data that is being sent to the private host from the outside network will be sent directly to the public NATs address port mapping. All data originating from the private host will also be assigned to the public NATs address port mapping. In Address Restricted Cones the private address and port of a host is mapped to an external pairing in the NAT similarly to Full Cone NATs. However, the Address Restricted Cones block any communications coming from the outside network to any port if the private host has not initated the communication from any of its ports. Port Restricted Cones work in a comparable way, but they extend blocking any communication coming from the outside network to a specific port. Symmetric NATs change the address port mappings for each request [21].

Out of the four, only the Full Cone typology can be used to bidirectionally communicate between hosts in private networks and public networks. However even with Full Cone NATs, the public host has to know the NAT port that is assigned to the private host's address. In P2P solutions this is often impossible as ISPs do not allow their clients to access routers to enable port forwarding. In total, NATs cause reachability issues to any host situated behind a NAT [21].

To get around this challenge, multiple methodologies for NAT traversal have been established. For the scope of this thesis the most important ones are Simple Traversal Utilities for NAT (STUN) [66], Traversal Using Relays around NAT (TURN) [62] and Interactive Connectivity Establishment (ICE) [39]. These methodologies are used to establish connections between hosts in the presence of NATs or firewalls.

STUN servers are used as a third-party server situated in the public network. When a host is situated behind a NAT it may send a request to a STUN stun server to discover the typology of the NAT that it is behind and the address-port combination used to access it. Modern STUN servers can also be used to check connectivity between two endpoints and to send keep-alive messages to maintain NAT bindings [66].

STUN servers can also be modified for hole punching. Hole punching can be used when two hosts are behind NATs [21]. Both hosts need to know the same STUN server in this case. When a host A wants to connect to host B they first send a request to a STUN server that records the IP address and port mappings of the hosts. Once the STUN server has received mappings for both hosts, it sends information of each other's public addresses to the hosts. When the hosts receive the information, they will send requests to each other. Sending the request opens a hole in the hosts' NATs to wait for a response. When the hosts' NATs receive each other's requests, they will process them as responses which causes them to accept the message. STUN servers work well for allowing communication between all NAT typologies except the Symmetric typology.

When hosts are situated behind Symmetric NATs, TURN servers are used to enable communication between hosts. Similarly to STUN servers, TURN servers are situated in the public network. Host send requests to TURN servers to open address port pairings called relayed addresses. Once the TURN server has accepted the pairing, it starts to relay messages between the hosts. TURN was designed to be used for media transportation. To support this TURN allows many peers to be assigned to one address port pairing.

ICE [39] can be used to combine the STUN and TURN techniques. When ICE is used, hosts behind NATs gather information from STUN and TURN servers to find all possible address candidates from the servers. After the addresses are gathered the hosts test the addresses with connectivity tests. If both hosts are behind NATs a signaling server may be required for the hosts to discover each other. The signaling server is simply used to relay the address information between the hosts. A simplified overview of the ICE technique with two STUNs, one TURN and a signaling server is shown in figure 2.4. In the figure the STUN, TURN and Signaling Server are placed in the public network while the hosts are placed in separate private networks.

When ICE is used hosts will be able to communicate through most NAT boxes. However, in some cases NAT traversal is not possible using ICE [21]. In these cases, other solutions need to be used for example the hole punching technique described before. Another downside with ICE is that establishing connections takes much longer. In an overlay

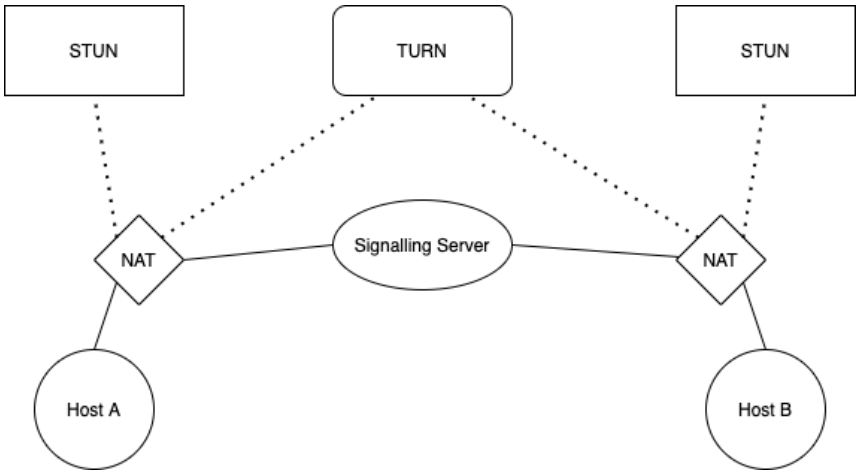

Figure 2.4: ICE technique architecture with STUN, TURN and ICE servers

network this creates problems with churn. Moreover, for decentralized networks the need for centralized servers to establish connectivity is not optimal. Despite this, ICE is used in the Streamr Network. There are solutions that allow hosts to establish P2P connections without centralized servers but these solutions are far more restricted to certain NAT implementations than ICE [25]. The main justification for the use of ICE in the Streamr Network comes from the use of WebRTC where ICE is part of the standardization. How the ICE protocol is implemented in the Streamr Network will be explained in chapter 4.

2.2.6 WebRTC Data Channels

WebRTC [87] is an API standard designed for forming P2P connections between browsers. WebRTC can be used for IPv4, IPv6 and dual-stack connections. Poor network quality in cases such as wireless connections and congested networks are also considered in the design [13]. The connections allow the browsers to directly send encrypted video, audio, and text to each other. WebRTC uses the STUN, TURN and ICE techniques for NAT traversal. For peers to discover each other and to exchange address information, a signaling server must be used. For media transport, Secure Real-Time Transport Protocol (SRTP) over DTLS is used. Text based data is sent over Data Channels which use SCTP over DTLS for transport. The use of DTLS is mandatory and one connection is used for both the media and text based data between peers. For the scope of this thesis, the Data Channels are more important as they are being used as the transportation mechanism in the Streamr Network.

Although the Data Channels support SCTP over DTLS over UDP by default, it is neces-

sary for the Data Channels to support TCP as well. Some firewalls may block all UDP communication, in which case TURN servers are often used with TCP connections. Negotiating a Data Channel connection ends up being quite complicated and time consuming as a result. The architecture for WebRTC Data Channels can be seen in figure 2.5

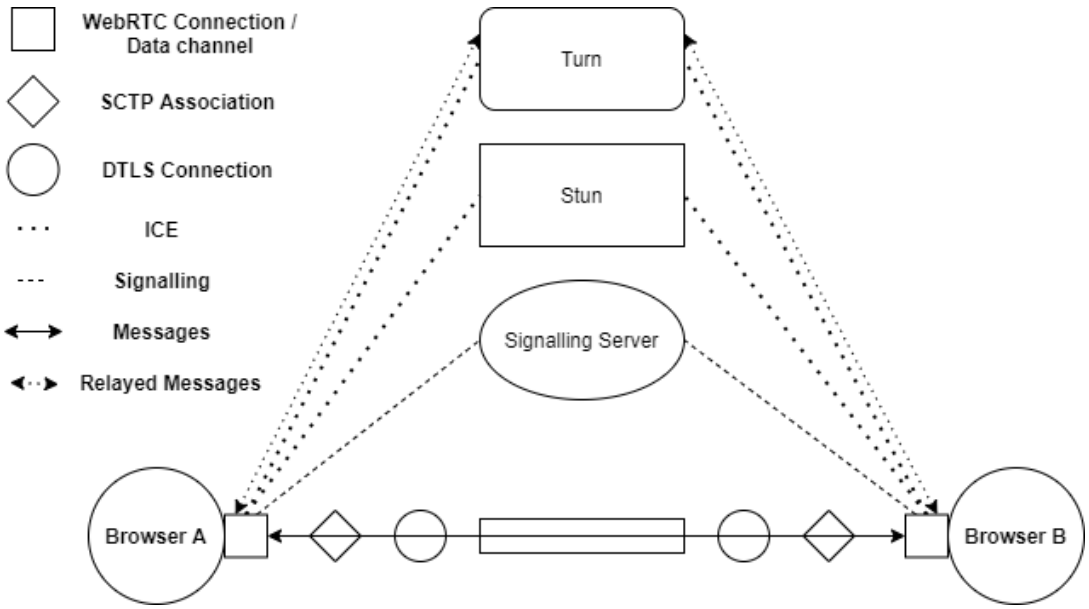

Figure 2.5: WebRTC Data Channel Architecture

When peers want to use an application that uses WebRTC they must first connect to a signaling server to discover peers. A common way to do this is by using WebSockets [88] but the WebRTC standard leaves the selected implementation for the signalling server up to the programmer. The signaling server only needs to be able to pass Session Description Protocol (SDP) [47] offer and answer messages between peers. The most important fields that are specified in the SDP messages for Data Channels are the used protocols and remote address and port information.

Once the connections to the signaling servers are established the peers may initialize PeerConnection objects. The peers will then start to gather ICE candidates from local routers and STUN and TURN servers. The PeerConnection object's createDataChannel method is called to initiate the connection. The gathered information is packaged as an Offer message using SDP and sent to a peer over the signalling server. The receiving peer will then initiate a PeerConnection object and start to gather ICE candidates if it has not done so already. It will then generate an SDP based Answer message with the ICE information and send it to the offering peer over the signaling server. Once the answer message is received and accepted by the offeror, ICE checks are performed. If they succeed,

a DTLS connection is initiated between the peers. Once the DTLS connection is open, the peers will negotiate the SCTP association for the Data Channel. Once the Data Channel is open the peers may begin to send data securely over the connection. If new Data Channels or media tracks are opened between the peers after a successful negotiation, the negotiation may be started from the ICE check if all necessary protocol information has been exchanged [87].

Messages sent through Data Channels are first sent through the SCTP association. SCTP is responsible for ensuring reliability and congestion control for the Data Channels. It is possible to configure the SCTP association's reliability and ordering functionalities, described in section 2.2.2. This is useful for modifying the channels for appropriate use cases. The SCTP association passes the message to the DTLS connection where it is encrypted and transported to the receiver. Each Peer Connection has one established DTLS connection used for both SCTP and SRTP traffic.

As all Data Channel messages go through a single SCTP association and DTLS connection, bottlenecks are bound to form. As the buffer and bandwidth resources can be shared between many Data Channels high throughput applications may experience issues. This can be a problem even using a single Data Channel in a Peer Connection as the data needs to pass from buffer to buffer. Especially, if large messages are sent through the connection [88]. To combat this challenge, WebRTC Data Channels have their own buffer which emits events when the buffer reaches high and low thresholds. Proper use of the buffer threshold events prevents applications from congesting the buffers. The thresholds can be modified to be appropriate for a specific application.

When comparing WebRTC to WebSocket it is clear that WebSocket is much more efficient in client-server use. However, if WebSocket is to be used in P2P overlays much of the overhead for forming connections between hosts behind NATs in WebRTC is required. The ICE architecture with the use of signaling servers or hole punching STUNs would be required in the modern internet. Regardless, WebRTC is expected to have worse performance for forming connections due to more RTTs in SCTP / DTLS than in TCP / TLS. Sending data through the Data Channels is expected to be slightly more costly as well. This stems from the data needing to be passed through Data Channel, SCTP and DTLS buffers.

2.3 P2P Overlay structures

There are many distinct types of overlay models. They are split into three categories: Cluster, Structured and Unstructured overlays. For the scope of this thesis the Unstructured overlays are the most important. However, it is important to also go through the methodologies used in Structured overlays as they are used in the Streamr Network's Trackers and related work. In this subsection, the most important methodologies for the Streamr Network and related work will be explored.

2.3.1 Distributed Hash Tables

Distributed Hash Tables (DHT) are commonly used data structures in overlay networks. Overlay networks based on DHTs are called structured networks. DHTs are great for distributed key-value pair storage. They support the insert(key, value), get(key) and delete(key) operations. All keys in the system need to be unique and hashing is used to give the node ids and the keys of the stored data unique values. Each node in the DHT overlay must keep a routing table to its neighbors and preferably to some more distant nodes in the DHT. There should be a network link to each direct neighbor in the routing table. The routing tables differ in their structure and thus performance in different DHTs. Although DHTs are not currently used in the Streamr Network outside of connecting trackers to one another, it is important to understand the many DHTs used in related work.

Chord [76] was originally introduced to provide deterministic locatability of data for P2P networks while remaining scalable and churn resistant. The biggest enabler for these qualities is Consistent Hashing [37]. Consistent Hashing solved churn related problems with traditional distributed hash tables. Traditionally, when a peer left or joined a DHT all the stored keys in the overlay had to be remapped. Consistent Hashing significantly reduced the remapping requirements during churn by making each node responsible for a range of hashed keys based on the node's and its successor's hashed ids. This reduces the complexity of joining and leaving the DHT from the traditional O(K) to O(K/n), where K is the number of keys and n is the number of nodes in the DHT. Consistent Hashing is so important for structured networks that is used in each of the systems described in this section.

Chord is the classic example of a DHT. Each node in Chord knows its success and prede-

cessor in the DHT. This gives Chord its signature ring shape. To avoid all routing in the DHT becoming linear in complexity, nodes in Chord maintain finger tables. Each of the m fingers is mapped to a deterministic hashed key in the DHT. The responsible node for the finger key will added to the finger table and a direct connection between the nodes will be formed. As the finger tables are constructed with a binary formula the routing in Chord ends up at O(log n) complexity.

Pastry [67] has a very similar performance and shape to Chord. However, it's routing differs from the finger table-based system of Chord. In Pastry routing is done by traversing the DHT based on the prefix of the hashed keys and node ids. This means that with each hop a message is passed to a numerically closer node in the DHT. The system allows each node to know small details of the DHT's structure globally while knowing more per each level in the prefix match. Similarly, to Chord's finger tables, the prefix-based routing and traversal is inherently logarithmic in complexity. Tapestry [4] is very similar to Pastry. It differs by using the suffixes of hashed keys for routing. The main difference in performance between the two comes from routing table maintenance, where Tapestry beats Pastry in complexity.

Content Addressable Network (CAN) [60] differs from the previous systems which are based on the ring topology. In CAN, nodes are placed on a virtual Cartesian coordinate space. Each node is assigned a zone in the space which can be split for new nodes joining the DHT. Routing from node to point is done by selecting a point P in the Cartesian space. The node will first ask its neighbors if their covered zones contain P. If this is not the case the node will start sending requests node-to-node between itself and P in space until the destination node is reached. One large benefit of CANs approach is that multiple dimensions can be added to the Cartesian space. Adding more dimensions improves the routing performance of the DHT significantly. However, each added dimension makes the maintenance of routing states and nodes joining and leaving churn more expensive. Each of these operations' complexities are O(2d), where d is the number of dimensions.

Kademlia [52] uses the XOR-metric for constructing its overlay topology. The use or XOR makes Kademlia's overlays reassemble binary trees. Many previous tree-based DHT systems had difficulties with maintaining the structure. They were not churn resistant and node failures could end up blocking out huge parts of the network. In Kademlia each node only maintains a small fraction of the global network in its routing table. The goal of the system is not to build a global binary tree. Instead, the routing tables collectively can reassemble a binary tree. This is important as it allows for the structure to actively

make use of redundant paths, which would not be the case in many tree-based systems.

In general, DHTs are highly decentralized, fault tolerant and scalable. Nodes in the DHTs can coordinate all functionalities without a need for a centralized entity. This makes DHTs one of the best available solutions for building decentralized networks. DHTs are also quite tolerant to nodes joining and leaving the network. However, this requires the use of Consistent Hashing which makes it possible to replicate data on redundant nodes. If Consistent Hashing is not used and a node leaves the overlay network, all the leaving node's data must be transmitted to the next node in the DHT. This can be quite a heavy procedure and leaving it undone leads to data losses. The scalability of the DHTs comes from the often constant time hash functions and logarithmic routing. These featuristics make it possible to scale DHT based solutions to millions of nodes without significant performance losses.

DHT based solutions do still have their flaws. The most obvious of which is the systems' vulnerability to different types of attacks. The most common of which are byzantine type Sybil Attacks [45]. In Sybil attacks, the attacker forms multiple false identities which can be generated from a single machine. Once the false identities join the DHT they can be used to alter the systems behavior. Commonly Sybil attacks are used to eclipse honest nodes from the rest of the network by only connecting to false identities. One commonly used way to prevent Sybil attacks is to use a reputation based system for the nodes [45]. However, even though multiple solutions have been proposed to the problem it appears that a full solution is yet to be found. This makes DHT based solutions quite risky for long term P2P overlay deployments.

2.3.2 Random Graphs

Overlays that are not based on DHTs are called unstructured overlays. In these systems the topology of the overlays is often based on random graphs. In random graphs the lack of consistent structure in the overlay topology requires the use of flooding to propagate messages to their recipients like in Gnutella v0.4. In flooding, each node in the overlay sends messages forward to each of their connections. This approach leads to many duplicates being passed around in the network. However, with proper filtering propagating duplicates can be avoided. It is also possible to add some structure to the topology to implement selective flooding similarly to Gnutella v0.7. This reduces the bandwidth use required by flooding.

For file systems such as Gnutella the performance overhead of flooding for lookups ends up being very costly at O(N) over the network. Any querying to the network requires the messages to be randomly propagated to find a node that holds the required data. The response must then be propagated back to the sender of the query via the same path unless a direct connection can be established based on the request's metadata. Nevertheless, random graphs are quite good for overlays that are used for file storage and sharing. As many nodes may be sharing a single file, a file can be downloaded quite fast from the network with proper partitioning.

Connectivity is a big challenge for random graphs especially when flooding is used for topology generation. Due to the random nature of the topology, achieving full connectivity of the graph is probabilistic. The probability increases based on the minimum number of connections per each node. It can be important to monitor the connectivity of unstructured graphs. The likelihood of network partitions and connectivity failures on the edges of the graph increase significantly in the presence of churn in unstructured networks [40].

Unstructured networks share many of the security flaws of structured networks. For example, it is possible to eclipse nodes from the network similarly to structured networks. This is possible as decentralized unstructured overlays often work based on peer sampling. Nodes only hold small views of the entire network for efficiency and share their views with neighboring nodes to find other nodes in the network. However, byzantine nodes can take advantage of this by only sharing knowledge of other byzantine nodes in the network [34]. By using this approach, the byzantine nodes can end up taking control of the network with relatively small numbers of nodes.

In random graphs, the hub attack is another huge security threat. In it byzantine nodes join the network and become hubs in the network by creating substantial amounts of random connections. Once enough nodes are connected to the byzantine hubs, they start blocking any traffic coming from their neighbors or disconnect. This type of an attack is quite dangerous as it can be used to take down the entire network with quite a small number of nodes. Only 20 nodes are required to take down a network of 1000 nodes [5].

2.3.3 Trackers

Trackers are used as centralized parties over unstructured overlay networks. They are often used to control how the nodes in the network should form their connections. The obvious flaw of with trackers is that on their own they break the decentralization of the network

and become single points of failure. Trackers also reduce the scalability of the overlay as they require all nodes to form one-to-one connections to them. Tracker controlled overlays were originally introduced in BitTorrent [36]. The use of trackers is beneficial to the overlay despite some downsides. The nodes are not responsible for controlling the overlay. The nodes simply connect to the tracker and wait for instructions on which nodes to connect to in the overlay. When nodes do not need control the overlay topology by gossiping the amount of control traffic is reduced by a significant amount. An additional benefit of the nodes not needing to participate in control traffic comes from the ability to make the overlays per topic and noiseless as described in subsection 2.1.2.

It is possible to decentralize the trackers by making them join a DHT. When this is done the trackers could be ran as part of the nodes [36]. However, by doing this most of the benefits of noiseless nodes would be lost. A better approach would be to make the trackers join a separate DHT overlay network where they can replicate their generated topologies for improved fault tolerance, availability, and scalability. A single tracker could be responsible for updating a single per-topic overlay topology on the DHT and could replicate its results to redundant locations. By doing this, nodes would connect to any tracker in the DHT, and the tracker would be able to find the appropriate overlay topology, insert the new node into it and give the node instructions on where to connect to.

It could be argued that centralized trackers could be used to improve the trust level of a P2P overlay. At least many corporations could trust P2P overlays if they could be in control of the topology or use a trusted party. On the other side of the coin, public overlay networks could face censorship under the control of centralized trackers. The ability to block content and nodes from an overlay network is never-the-less a useful feature in many cases. It can be used to improve the security of the network by denying nefarious nodes access to the network and to block illegal content from the network. By connecting the trackers to a publicly accessible DHT, the benefits of decentralization would be gained. However, most of the security benefits provided by trackers would be lost. Once the trackers are connected to a public DHT they become vulnerable to all the downsides of structured networks. As the Trackers are responsible for controlling the connections of the nodes in the network, taking control of them could cause network wide damage.

The efficiency of P2P overlays when using trackers is undeniable. Trackers significantly reduce the processing power and bandwidth that peers in overlays require. In cases such as IoT this is beneficial. Placing P2P publish-subscribe brokers near IoT devices or in their control devices could be a great way to improve the scalability, availability, and fault toler-

ance of IoT. Especially, if the brokers would be deployed on low-capacity devices. Trackers also trivialize adding optimizations to the overlay topologies. Handling algorithms that form network topologies is much simpler in a centralized than distributed fashion, not to mention deploying changes to existing environments. The centralized control also comes with the ability to add a level of reliable structure to the overlay topology which is not possible in trackerless unstructured overlays. The connectivity concerns of unstructured overlays can also be forgotten as the tracker always holds a view of the overlay topology.

2.4 Ethereum

Blockchains are decentralized systems that are used to store records called blocks into chains [55]. Each block holds a cryptographically hashed key of the previous block in the chain, timestamp, a random value for validating the block and transaction data represented as a Merkle Tree. Merkle trees [78] are cryptographic binary trees that store their values in their leaf nodes. Each parent nodes value is a combination of the hashes of its child nodes values. This means that altering any value in the struct requires a change in all of the parent values in the tree. This makes the structure resistant against altering of records and thus great for distributed ledgers such as blockchains.

Blockchains are based on having multiple different parties that do not need to trust each other storing their own copies of the entire transaction history. All transactions sent to the blockchain are hashed cryptographically and nodes called miners attempt different keys to try to solve the puzzle. The first node to solve the puzzle adds the record block to its chain and sends it to the rest of the nodes in the system. This requires a P2P overlay-based system to be used as the backbone for message propagation in blockchains. The nodes that solve puzzles and hold copies of the blockchain are often called miners and can be paid small prices for being the first one to complete a puzzle.

Ethereum [12] is a blockchain that can be used to run deterministic code in a decentralized manner. A Turing-complete programming language is provided with the Ethereum blockchain. This makes it possible to create decentralized applications that run on the Ethereum Virtual Machine (EVM). Applications running in the EVM are called Smart contracts. Running transactions and smart contracts in Ethereum costs a small amount of Ethereum's tokens called Eth. The transaction costs are often referred to as gas. The gas prices are market driven and are used to pay the miners for solving puzzles.

Smart contracts are faced with some limitations due to the requirements for deterministic

operations. All state transitions on the EVM need to be deterministic [31]. If computations can have random results the recorded blocks may end up having inconsistent values and the transaction becomes useless. This means that features such as concurrency and loading files from disk or the internet is not possible. As an additional security measure, a smart contract should not be able to change values of another smart contract. Once deployed smart contracts code cannot be altered. Deployed smart contracts stay on chain forever and can only be updated by deploying a new contract with a new address. Despite these drawbacks Smart contracts can be used to create decentralized authentication and authorization layers, reputation systems and to store small amounts of immutable data [12]. For example, the tracker addresses of the Streamr Network can be found in a smart contract to bootstrap new nodes into the overlay.

3 Requirements and motivations for the Streamr Network

As a P2P topic-based publish-subscribe system, the Streamr Network has a clear set of motivations and requirements laid out for its design. This chapter goes over the requirements and motivations for the system. To do this, example use case scenarios are provided, the system requirements are listed and related work in the academic and real-world use are explored. The example scenarios and expected requirements that the Streamr Network should support are laid out in the initial whitepaper of the Streamr project [58].

3.1 Example use cases

The example scenarios for the Streamr Network are focused on large scale open data. This means that the network should leverage the high scalability and availability of decentralized publish-subscribe networks. The specific qualitative requirements for the network are explored in the next subsection. In this subsection, specific example use cases for the Streamr Network are discussed. These examples set the motivations for the Streamr Network as a real-time data marketplace [58].

The network should be able to handle real-time data of events in the stock market. Events of new offers, bids and trades could be published to the network in a topic per stock basis. In this example anonymity with ability to verify digital identity, low latency and high availability are important.

The network should also be able to support machine to machine (M2M) data. An example of this is a network of connected cars. The cars would publish data of their speed, acceleration, location and any notifications and warnings to topics based on geographical location. The cars should be able to connect to multiple streams to account for boundary areas between geographical topics. The connected cars would also receive all data that the other cars that are publishing to topic. This would allow drivers or cars to react to alerts originating from nearby traffic. For example, nearby accident alerts could trigger speed limits or emergency braking in worst case scenarios. Self-driving cars would also receive an additional layer to their sensing capabilities by being able to interpret data from nearby

cars.

Smart city open data is a wider use case that the network should support. For example, individuals living in cities could publish data of their air quality, pollution, and weather monitoring sensors to topics. Wide scale adoption of such a system could be used as a backbone for new types of decentralized weather services. Another smart city example is the real-time data generated by sensors in public transportation vehicles. The vehicles transmit data of their identities, location, speed etc. at a high frequency rate. The challenges of adapting centralized brokers to thousands or millions of members in such networks are quite apparent. One of the main challenges being the maintenance of connections from a small number of centralized servers. The Streamr Network attempts to solve the problems of scalability and fault tolerance in centralized solutions with decentralization.

3.2 Requirements

The requirements of the Streamr Network are laid out in [70]. They are based on the original whitepaper for the Streamr Project and the example scenarios for smart cities and connected cars. The 11 main requirements are explained in detail in The Streamr Network whitepaper. Here they will be explained more briefly to give context to the related work in the next section and the design in chapter 4.

1. Scalability

   The goal is for the design to scale without limit. Moreover, the quality of service should not suffer as increasing numbers of nodes are added to the network. The need for this requirement become apparent in all the example use cases. If peers need to send all events with unicast to other interested peers, they will not be able to keep up with the associated uplink costs as more nodes join the network. Instead, the load of the participating peers in the network should be kept low and constant.

2. Decentralization

   The main goals for this requirement in the Streamr Network are the absence of hot spots and central points of failure. The network should remain functional even if larger parties, for example cities in the open smart city data scenario, leave the network. The importance of decentralization can also be seen in M2M data scenarios where it is important for devices to be able to communicate despite differences in manufacturers.

3. Low and Predictable Latency

   When messages are sent over the network the measured end-to-end delays should be close to unicast message delays. This is important for cases such as the connected cars scenario. The cars should receive data within reasonable time periods to ensure that the data being sent over the network is useful. In addition, any critical alerts sent over the network should be received by important parties before accidents happen. P2P Overlay networks are bound to have an increased latency for message propagation when compared to centralized systems. However, the topology of the overlay can be tuned and designed for low mean message propagation delays.

4. Optimization for Small Payloads

   The network should be tuned for sending large numbers of small messages instead of sending small numbers of large messages. The root for this requirement comes from the system's design towards IoT, where most M2M message payloads are small.

5. Bandwidth Efficiency

   There should not be large volumes of unnecessary messages in the network. Importantly, nodes should not receive large numbers of duplicates per message. In low resource IoT devices, sending and receiving many duplicate messages could exhaust CPU and battery resources.

6. Message Completeness

   The nodes should be able to receive messages that they are interested in with a high probability. As a precaution, it is important for nodes to have mechanisms to detect lost messages. When lost messages and gaps are detected the nodes should be able to ask nearby nodes for retransmissions.

7. Churn Tolerance

   The network should be able to tolerate nodes joining and leaving the network at a high rate. The quality of service of the network should not be hampered during churn. For example, in the connected cars example vehicles would be transferring between zones at a fast rate.

8. Zero Noise

   The noisiness of networks has already been discussed earlier in the thesis. Zero noise implies that nodes should not receive any traffic that they are not interested in. In Streamr's case the goal is that even unnecessary control traffic should be

avoided. This means that the peers in the network should not act as relays by default. This makes structured networks a difficult solution for the peer overlay of Streamr Network. The highly decentralized nature of structured networks makes all peers quite noisy as they need to process and forward control traffic. In worst cases, peers may end up acting as a hot spots for control traffic that they are not interested as will be seen in section 3.3.2.

9. Fairness

   Nodes should not be able to optimize their own place in the overlay topology for their own gains. The importance of this requirement comes apparent in the financial data scenario. It can be unfair for other subscribing peers if one peer finds a placement which makes it receive data faster or slows down the reception of data for any other nodes.

10. Attack Resilience

    The goal of the Streamr Network is to be resilient against all types of known attacks. This is where duplicate messages become useful once again as they provide protection against censorship from neighboring peers. The requirement makes it quite difficult to design the network based on truly decentralized patterns.

11. Simplicity

    The design and implementation of network should be kept simple and easy to debug. This makes the system more reliable.

The first two requirements for the Streamr Network limit the design to a P2P overlay solution as modern Client-server solutions cannot fulfill both requirements. These two requirements are the most important as they are the most limiting ones for the system's design. The rest of the requirements must be accommodated to the overlay network model. Designing the system with the requirements in mind, ends up being quite a balancing act.

Optimizing for small payloads (4) and *bandwidth efficiency (5)* come with the expectation of low control traffic. This makes random graph type topologies more favorable than tree type topologies [44]. The unstructured nature of random graphs significantly reduces the amount of required control traffic even compared to simpler structures such as binary trees. However, duplicate messages are also quite unavoidable in random graphs [49] which increases bandwidth requirements. Despite this, duplicate messages imply redundant paths for messages which makes it more likely for all nodes to receive them in all cases. This

makes the system more reliable, robust and churn tolerant. Overall, this analysis leads to graphs being the favored for the topic overlay topology of the Streamr Network.

The requirements for *Zero Noise (8)* and *Attack resilience (10)* make structured networks unlikely candidates for the Streamr Network. Nodes often end up acting as hot spots for control traffic that they are not interested in many as will be seen in section 3.3.2. DHT based solutions are also quite vulnerable to several types of attacks as described in section 2.3.1.

Despite many of the known issues with structured solutions, unstructured systems have their own flaws in terms of the listed requirements. For example, the *attack resilience (10)* requirement is hard to fulfill. This is the case especially if the system is fully decentralized as described in section 2.3.2. Although, there are proposed solutions to fix these problems such as [9], they often end up being unproven in practice and overly complicated, going against the requirement for *simplicity (11)*. Similarly to DHT based solutions, the requirement for *Zero Noise (8)* is difficult for decentralized unstructured overlays as will be seen in section 3.3.1. The requirement for *fairness (9)* is also quite challenging. This is due to the fact nodes can optimize their position in the overlay topology in unstructured networks where peer-sampling is used.

Based on the related work in the next section, it will become quite apparent that tracker-based solutions can be used to alleviate the challenges of balancing the requirements. However, the use of trackers does come with adverse effects to the requirements for *scalability (1)* and *decentralization (1)*. Despite this from the point of view of the nodes the *bandwidth efficiency (5)* and *Zero Noise(8)* are improved as there is no need to propagate control traffic, *fairness (9)* is improved as the nodes cannot easily optimize their positions, *simplicity (11)* is improved as the control traffic communication between the tracker and node is much simpler to implement and debug than in decentralized solutions and the requirement for *low and predictable latency* gets an improvement as the graph states constructed by trackers in combination with node-to-node RTT data can be used to predict latencies in the network.

3.3 Related Work

To support the design of the Streamr Network based on its requirements it is important to understand its related work. Many academic decentralized publish-subscribe systems have been proposed. Despite this and their many advantages, they have not seen widespread

real-world adoption. The main reason wide real-world adoption for these systems has not happened are the security problems related to attack vulnerabilities. These security faults have already been discussed and thus the systems' individual attack vulnerabilities will not be explored in this section. The security faults are quite similar in most structured and unstructured networks respectively. The focus of this section will be on some of the implementation details of the related work and how their design fits the requirements for the Streamr Network. The DHTs used as the basis for the structured systems presented in this section were described in section 2.3.1.

3.3.1 Academic examples of unstructured overlay systems

Tera is a topic-based decentralized publish-subscribe system presented in [6]. Tera uses two different levels of unstructured networks to form its overlay. The first overlay is global and is used for Outer-Cluster Routing. Each node in the global overlay maintains a random set of connections to peers. All peers advertise the topics they are interested into their neighbors and maintain a topic-to-node mapping called the subscription table. The subscription tables contents are being advertised to a random set of nodes in the global overlay periodically. The global overlay is also responsible for arranging the second level overlays which are the per-topic overlays of the system. This part of the global overlay's responsibility is called the Overlay Management Protocol. When a node wishes to subscribe to a topic it needs to add itself to subscription tables and asks the Overlay Management Protocol to join the topic overlay. Joining the topic overlay requires the joining node to find a node that has already joined the topic overlay. If such a node is not found the node must create the topic. Unsubscribing is then handled by the node leaving the subscription tables and asking the Overlay Management Protocol to leave the topic overlay.

The per-topic overlays of Tera are responsible for Interest Clustering and Inner-Cluster Dissemination in the system. Each topic overlay only contains nodes that are interested in events published to the topic. This ensures that nodes do not need to handle noise once an event has entered the topic overlay. However, when events are published from outside the network, the message can end up being first received by a node that is not subscribed to the topic. In these cases, the nodes will end up working as relays to forward the data to the topic overlay. This introduces some unnecessary noisiness to the system. However, the noise should only be present for the first published events. Once a node that's connected to the topic overlay receives the event it will send an acknowledgement

back to the outside publisher. If NATs or firewalls do not prevent forming the connection to the subscriber, the noisiness can be avoided on later events. The high node degree commonly present in the topic overlays of Tera can cause problems for nodes that are subscribed to multiple topics. For example, the link bandwidth of a node could become a limit when the system needs to propagate traffic in high throughput topics. Moreover, the global control overlay requires additional connections to be maintained. These factors do not fit the requirements for *scalability (1)* and *bandwidth efficiency (5)* very well.

Spidercast [17] is a topic-based decentralized publish-subscribe system that uses a single global overlay for both control and content traffic. Where Tera focuses on Inner-Cluster Dissemination and Outer-Cluster Routing, Spidercast focuses on Interest Clustering and reducing the number of maintained connections per node. Nodes select their neighbor with bias towards shared interest in topics. The nodes attempt to keep the topics at least K-connected, where K is the goal number of connections per node in a topic.

Similarly to Tera, Spidercast has two separate overlay protocols used for subscription management and overlay construction. However, both of these protocols are used to maintain one global overlay. The subscription management protocol is used by nodes to hold an interest view of other nodes in the system. The authors showed that each node only needs to know 5% of random nodes in the system for it to maintain full topic connectivity.

The second overlay protocol in Spidercast is called the overlay construction and maintenance protocol. Initially when a node subscribes to a topic it will use interest views to find nodes to connect to. The subscribing node will then form connections to the found nodes up to a certain limit that is higher than K. Once the limit is reached the node will begin to disconnect itself from over-covering neighboring nodes. To avoid reducing the neighboring nodes' connections below K, each node maintains the neighboring nodes' topic node degrees. After the initial construction, neighbor maintenance is started. During maintenance several routines are repeated to ensure that K-connectivity is maintained for the topics and that events are handled and propagated over the topic networks. The construction and maintenance protocol is also used to limit the number of connections that the nodes are not interested in. The authors showed through simulation that the event propagation in the system's topics is noiseless. However, the Outer-Cluster Routing in Spidercast is still very noisy in terms of control traffic. Thus, it does not fit the requirements for *bandwidth efficiency (5)* and *Zero Noise (8)*.

PolderCast [71] aims at similar goals to Spidercast with full topic-connectivity while main-

taining a low node degree. Similarly to Tera, PolderCast uses per-topic overlays for Inner-Cluster Dissemination and a global overlay for Outer-Cluster Routing. However, PolderCast has design goals that the previous system did not explicitly have. Two goals that the previous unstructured systems do not have are low latency event dissemination, low numbers of duplicate events and 100% event publication delivery guarantees under zero churn. The latter of these design goals is not achieved in Tera or Spidercast due to partial views of interested nodes. This approach combined with random graph overlays can lead to probabilistic event delivery and overlay topic partitions.

Similarly to the previous unstructured solutions, PolderCast uses two different protocols for finding interested nodes and forming topic overlay connections. While the previous solutions introduced their own implementations for these protocols PolderCast uses ready-made solutions. Cyclon is used as a peer-sampling system for Interest Clustering, Vicinity is used to construct the unstructured overlay connections for topics and Rings is used to generate ring topologies for the topics overlays.

When a node subscribes to a topic in PolderCast, it will first use Cyclon to find peers already in the topic overlay similarly to Tera. Vicinity is then used to connect to the found nodes. This process is similar to Spidercast with key exception. Vicinity attempts to find nodes with a bias towards nodes that are connected to multiple shared topics. Even though the topic overlays are formed as separate application logic overlays with K node degree, the underlying connections between the nodes can be shared. The Rings system will then form ring topologies with the sampled nodes based on node ids. The use of ring topologies gives the topic-based overlays some level of structure which improves the delivery guarantees of the system. To improve the event dissemination latencies, random connections are added between the nodes across the ring. Even though PolderCast improves the message delivery guarantees, churn resistance and reduces duplicate bandwidth compared to the previous systems, the system does not benefit the message propagation latencies. This is a result of the ring topologies being formed based on node ids and not latencies between the nodes. However, this could be improved by prioritizing latencies when forming random connections across the rings. This does not fit with the requirement for *low and predictable latency (3)* of the Streamr Network. Moreover, the system has similar problems as Spidercast in terms of the noisiness of Outer-Cluster Routing.

Each of these systems come with the security flaws of decentralized unstructured networks as described in section 2.3.2. The main security flaw in these systems is the gossip-based peer-sampling [34]. The nodes only know a partial view of the entire network in each of

these solutions. The views are kept up to date by periodically exchanging information with neighbors. This makes the networks particularly vulnerable to eclipse attacks as byzantine nodes can select to advertise their views to only contain other byzantine nodes.

3.3.2 Academic examples of structured overlay systems

Bayeux [95] is an early structured decentralized topic-based publish-subscribe system. Bayeux is based on the Tapestry DHT described in section 2.3.1. The DHT is used for Outer-Cluster Routing to find the root nodes for each topic. Per-topic multicast trees originate from the root nodes. These trees are maintained separately to the DHT system. However, when events are being propagated between nodes in the topics, the nodes in the multicast trees need to be found based on node ids in the DHT system. This means that the nodes in the system end up relaying loads of control and event traffic that the node itself is not interested in. These properties make the system very noisy. The noisiness of the Inner Cluster Dissemination strategy could be improved in the system by simply using underlying IP addresses when constructing the multicast trees. This would avoid the overhead of DHT based routing for message propagation.

Another early academic system is Scribe [14] which is based on the Pastry DHT. The system is similar to Bayeux as it constructs per-topic overlays and uses the DHT for routing messages to the nodes in its multicast trees. However, the noisiness in Scribe is even worse than Bayeux as uninterested nodes can be set as root nodes for the multicast trees.

Application-level multicast using content-addressable networks [61] is a topic-based decentralized publish-subscribe network. It is based on the CAN DHT which is apparent in its name. A single global DHT is used for Outer-Cluster Routing in the network to find bootstrap nodes for multicast groups. Each multicast group is then maintained by its own CAN DHT making the system quite similar to Tera. Propagating messages can then be implemented with simple flooding or more complicated multicast techniques. As the Outer-Cluster Routing is done based on node ids in the global DHT, uninterested nodes may end up being bootstrap nodes for topics. However, compared to the previous structured systems the Inner-cluster Dissemination is noiseless.

All DHT based systems have a problem with the creation of control traffic hotspots. This is inherent in DHT routing because of its decentralized nature. It is possible to avoid this by making the peers in DHTs directly connect to more peers. However, this has unwanted

effects for the scalability and churn tolerance of the system. Moreover, without proper design the peers will end up relaying unnecessary events as well. Finally, as described in section 2.3.1, DHT systems have their security flaws. The attack vulnerabilities of structured networks are the biggest reason DHT based public systems have not seen adoption in the real world.

All in all, as the Streamr Network wants to limit control traffic, make its nodes as noiseless as possible and be attack tolerant, DHT based solutions can be ruled off even though they offer the best scalability and decentralization.

3.3.3 Examples of related real-world deployments

As the previously described academic systems have not seen any real-world adoption, it is necessary to look at how successful P2P solutions have been deployed in the real world. The key is to find examples that reassemble topic-based publish-subscribe systems. As such systems have seen limited real-world adoption related systems that reassemble publish-subscribe systems should be explored. The first and last examples, SWAN and Matrix are rare examples of P2P publish-subscribe systems that have seen real world use. The second example is BitTorrent which has some similarities to topic based publish-subscribe systems. On top of these systems, blockchains have become widely deployed and used examples of decentralized overlay networks. The Ethereum blockchain has a role in the Streamr Network's architecture and was described in section 2.4

SWAN is [32] an example of a commercial decentralized publish-subscribe system. It is based on an unstructured overlay network using k-regular random graphs. Each topic in SWAN builds its own overlay and all events published to a topic are flooded all of the topics subscribers similarly to Tera. In SWAN a list containing topic members is distributed to each new peer. The list is used to find existing members of topics when a peer wishes to become a subscriber. Although SWAN is no longer used and its Outer-Cluster Routing strategy was inefficient, its use of random regular graphs for Interest Clustering and Inner-Cluster Dissemination is remarkably similar to the Streamr Network.

When a new peer joins a topic in SWAN it performs k/2 random walks over the network to choose k/2 connections between peers. It will then instruct the two peers of each connection to disconnect from one another and will then form connections to the disconnected peers. When a peer leaves the network gracefully, it will instruct all its k connections to form connections between each other. This can be seen as a reverse interaction to the

peers joining the network. When a peer leaves the network without a warning, the neighboring peers will attempt to find other nodes with less than k connections in the network by broadcasting messages. This procedure is used to try to maintain full connectivity of the network. SWAN could be seen to fulfill many of the requirements for the Streamr Network. However, maintaining the lists of peers in topics causes heavy control traffic in the network. Moreover, the network is vulnerable to similar attacks as Tera.

BitTorrent [36] is a decentralized file sharing system. BitTorrent users publish torrents which are collections of downloadable files. Interested users can then access the files by contacting trackers that have been listed in .torrent files. The .torrent files can often be found and downloaded from indexing websites. The trackers will instruct peers to join the torrent overlay to download pieces of the files from peers that already have local copies of the pieces. The constructed overlays are high node degree random graphs. BitTorrent has a wide variety of strategies for uploading, downloading, and forming overlay connections. The strategies are used to improve the quality of service of the system. However, describing the strategies is out of the scope for this thesis. What is interesting is the parallelism between BitTorrent and decentralized topic-based publish-subscribe systems.

The trackers in BitTorrent could be seen as a party that constructs per-topic overlays. Based on the trackers instructions the peers form an overlay with a high node degree of up to 55 [80]. The peers then use HAVE messages to inform neighboring peers of pieces of torrent data that they already hold. The neighboring peers can then request to download the pieces from the peer using different strategies. Using the HAVE messages comes with an expectation that the downloadable pieces are significantly larger than the control messages. If this is not the case the system will be flooded with control messages. The requirement of the Streamr Network for optimization for small messages and low control traffic makes such pull-based systems undesirable. However, by forming the per-topic overlays based on tracker instructions the peers in the system become entirely noiseless. They will no longer receive or process any noisy Outer-Cluster Routing, Interest Clustering, or Inner-Cluster Dissemination traffic. The system has the disadvantage of requiring centralized trackers that can be seen as single points of failures. The trackers thus reduce the decentralization of the system and come with some new attack angles to the network. However, the high node degree of BitTorrent comes with problems towards *scalability (1)* and *bandwidth efficiency (5)* in the requirements for the Streamr Network.

Trackerless BitTorrent [46] replaces the centralized trackers with a decentralized tracker system built on top of the Kademlia DHT. All BitTorrent clients run a tracker peer in

the system. This significantly increases the *decentralization (2)* and *scalability (1)* of the system. However, the peers may now need to route and can become hot spots for uninteresting control data violating the requirement of *Zero Noise (8)*. In terms of the *attack resilience (10)* and *simplicity (11)*, the removal of the single point of failure comes with the cost of being vulnerable to known attacks against DHTs and increased complexity of the system.

Matrix [19] is an example of a decentralized topic based publish-subscribe system that is in use today. It is built as a multipurpose network for real-time data. The topics in Matrix are called rooms and have been used for the IoT, Voice and video calls over IP and instant messaging. The Matrix network is formed by Homeservers that form a server-to-server network. The Homeservers are responsible for running Domain Name Server (DNS) directories that are used for mapping the topics. The directories serve a similar function as the trackers in BitTorrent. Clients can subscribe to the rooms by connecting to the server-to-server network. However, the clients do not join the fully connected mesh overlay that is formed in Matrix. Instead, they connect to a single Homeserver in the overlay. These two qualities significantly restrict the *scalability (1)* and *bandwidth efficiency (5)* of the Matrix overlay network as the number of connections and the volume of data that is pushed through them increases linearly.

4 Design and Implementations of the Streamr Network

When building any distributed system there are bound to be challenges and trade-offs. The CAP-theorem [11] states that it is impossible for any distributed data store to provide more than two out of three of the following guarantees: Consistency, availability, and partition tolerance. This characteristic of distributed data stores is also relevant for publish-subscribe networks and overlay networks in general. The design of the Streamr Network is not looked at directly through the lens of the CAP theorem in the thesis. However, the CAP-theorem is a fitting example of what kinds of design trade-offs are necessary in distributed systems.

The fallacies of distributed computing [79] is a commonly used warning against building distributed systems. Many of the fallacies can be seen to be less relevant today as most distributed systems are deployed in cluster setups in cloud environments. In [79] it is argued that the fallacies originally proposed in the 1990s are still relevant today in IoT systems. The bad connections, variance in end-to-end latencies (jitter) and device heterogeneity often present in IoT setups are presented as the main argument for the presented case. It is quite apparent that the same case can be made for decentralized P2P systems and thus the fallacies need to be taken into account when implementing decentralized systems.

The combination of these two theorems leads to several tradeoffs in the design and challenges in the implementation of any distributed system. This chapter goes into detail on what design tradeoffs have been made in terms of the requirements of the Streamr Network and the differences in implementation details between the WebRTC and WebSocket versions.

4.1 System overview

The Streamr Network is a decentralised topic based publish-subscribe system. The topics in Streamr are referred to as streams. Similarly to systems such as Tera and PolderCast, the Streamr Network constructs per-stream overlays. Each stream is operated by a BitTorrent-like tracker. Multiple trackers can exist in the system to balance the load of

overlay construction and maintenance. By default, the streams are balanced between the trackers with the use of Consistent Hashing. However, the owners of streams can assign specific trackers or their own custom trackers to maintain the overlay. The trackers are registered in an Ethereum Smart Contract that is used for bootstrapping. This makes it possible to setup public instances of the Stream Network with a default Smart Contract or for communities or consortia to setup private instances of the Network with permissioning using custom Smart Contracts.

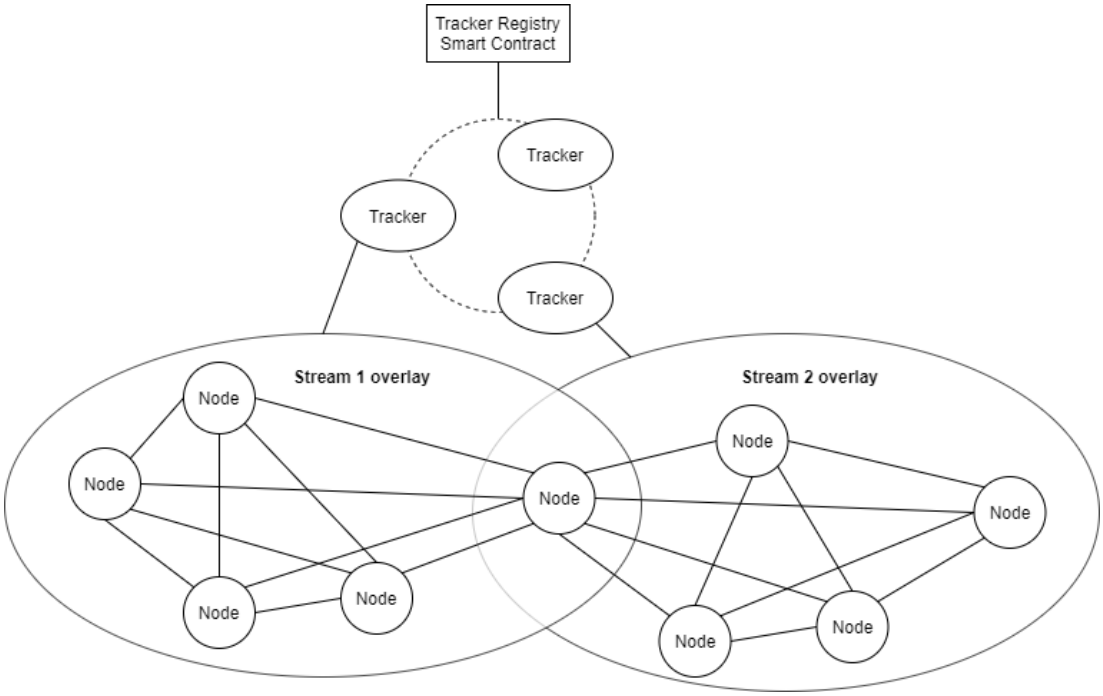

Figure 4.1: High level overview of the Streamr Network with three Trackers and two Streams

A high-level presentation of the Streamr Network can be seen in Figure 4.1. For simplicity any one-to-many connections have been simplified as a single line. In reality, each node in a stream overlay maintains a WebSocket connection to the responsible tracker and each tracker's address is stored in the Tracker Registry Smart Contract. In the figure, both streams have a node degree of four with a total of 5 nodes in the overlay. The figure depicts how a single node would connect to multiple streams at once. In the worst case, a multi-stream node may need to connect to each instructed node in a separate overlay. However, when two nodes subscribe to the same two streams it is possible for them to share the underlying connection between the streams. This reduces the connection maintenance overhead of a node taking part in multiple streams. Currently, the trackers do not implement any sort of redundancy. This can cause the trackers to become single points of failure. However, as described in section 2.3.1, it is possible to implement redundancy

quite easily with the use of consistent hashing in the future.

When a new node wishes to join a stream, it must first access a set of trackers from the Smart Contract. It will then connect to the set of trackers and find the tracker that is responsible for the stream. Next, it will inform the responsible tracker that it wishes to join the stream and will receive instructions from the tracker on which nodes to connect to in the stream's overlay. Upon receiving the instructions, the node will attempt to fulfill them by connecting to the instructed nodes. Once the instruction handling is completed, the node will inform the tracker of its current stream state. If any expected neighboring nodes are missing from the sent stream status message, the tracker will interpret that the forming of the underlying connection has failed. The tracker will then find a new random node to replace the failed connection. The regeneration process will be repeated until the node has reached k-connectivity. The node can begin to propagate messages to and from neighboring nodes once the first instruction is completed. It may also publish new message to the stream if it has the permission to do so.

4.2 Overlay Structure

The per-stream overlays of the Streamr Network are formed as k-regular random graphs. The goal is to maintain a low number of connections between each node. By default, the number of overlay connections per node (k) is set to four. In practise, each node will try to form these connections using the WebSocket or WebRTC protocols. Any duplicate connections between two nodes will be collapsed into a single underlying connection. Similar action was taken in SpiderCast and the approach was shown to be noiseless, while maintaining low node degrees in the overlays [17].

4.2.1 Trackers and overlay maintenance

The main responsibilities of the trackers in the Streamr Network are peer discovery and overlay construction. These two fall into the two categories of Outer-Cluster Routing and Interest Clustering respectively. As the trackers in the Streamr Network are responsible for two traffic confinement strategies, the trackers in Streamr are bound to hold more responsibility than Matrix's directory servers and BitTorrent's trackers. In Matrix, the directory servers can be seen to be responsible for both as well. However, the directory servers inform the nodes of all other interested nodes in the topic and the nodes connect to

each of them. This makes the Interest Clustering strategy much simpler than the Streamr Network's.

In BitTorrent, the trackers send a set of random interested nodes back to a requesting node. Based on this set, the node will then select who it wishes to connect to. BitTorrent's approach can be described as pull based, where a client asks the server to perform a certain task. The tracker in Streamr is better described as push based, as the tracker instruct the nodes directly on who to connect to or to disconnect from. Although this approach significantly reduces the decentralization of the network, it does come with its benefits. Eclipse attacks are much more difficult to perform as there is little room for nodes to optimize their position in the network topology. Moreover, the task of optimizing the overlays for *low and predictable latency (3)* and *bandwidth efficiency (5)* is more trivial. The nodes can send RTT values of their neighboring connections back to the trackers. The trackers can use this information to create locality aware stream overlays. Such solutions require much more coordination with a pull-based tracker and even more so in fully decentralized solutions. Intentional locality awareness has not yet been introduced to the Streamr Network, but will likely be explored in the future.

The Streamr Network's overlay construction algorithm is inspired by the Steger and Wormald algorithm [75]. The algorithm is designed to build regular random graphs quickly. To generate a random graph with node degree d and number of nodes n, the first step is to create a set U that contains all of the d unpaired edges of all n nodes as n * d points. There must also be a set of n groups V that contains n * d points in total for pairing the points. After initiation the algorithm proceeds as follows:

1. Two random points i and j from the set U are selected. If the points i and j are suitable based on any given set of conditions, they will be paired in the set of groups and removed from the set U. This process is repeated until suitable pairs can no longer be found.

2. A graph G is created based on the set of groups V. An edge between the ath and bth vertices will be formed only if a paired point exists in both the ath and bth groups. If the completed graph is d-regular it will be outputted, if not the process is started from the beginning.

In P2P use, a significant issue with the Steger and Wormald algorithm is that it is designed for generating static graphs. This means that the algorithm must generate an entirely new

graph when a node leaves or joins the system. In the dynamic nature of P2P networks this is unacceptable. Therefore, a new regular random graph generation algorithm was designed for the trackers in the Streamr Network. The algorithm is executed whenever a node joins or leaves the network. This includes ungraceful leaves, detected by neighboring nodes. In the following description of the algorithm, V is used to denote the set of nodes that still have unused connection slots in the current graph:

1. Choose two random nodes i and j in V, and if they are suitable based on a set of conditions, connect the two nodes and delete them from V if all of their connection slots are filled. This step is repeated as long as suitable pairs can be found.

2. Delete all nodes from V that have less than two unused slots.

3. For node $v \in V$, pick two random nodes a and b that are suitable to be connected to v. Next disconnect a and b from one another and connect the new unused slots to v. If v has less than 2 unused unused slots left, delete it from V. This step is repeated as long there are nodes in V.

This approach has some similarities to that of SWAN's overlay construction algorithm which was described in section 3.3.3. The interpose joining strategy of SWAN's algorithm is similar to the 3rd step in Streamr's algorithm. Both find a random node with less than d connections and place the node between two already connected nodes. Moreover, the 1st step of the Streamr tracker algorithm is similar to SWAN's ungraceful disconnection / maintenance strategy. In both cases, nodes with less than d connections are paired. The similarities between the systems indicate that the research on regular random graphs used to predict SWAN's performance [90] can be used to predict that of the Streamr Network.

Due to the tracker being responsible for Outer-Cluster Routing and Interest Clustering, the overlay construction control data in the Streamr Network is entirely noiseless. This is ensured by the control data flow being restricted to WebSocket communication between the nodes and tracker. Overall, with per-stream overlays and tracker-to-node control data, Outer-Cluster Routing, Interest Clustering and Inner-Cluster Dissemination are all noiseless. This means that the Streamr Network completely fulfills the requirement for *zero noise* by design. This comes with significant benefits to *bandwidth efficiency (5)* and *Simplicity (11)* as the broker nodes should not receive any uninteresting messages.

4.2.2 Nodes and message propagation

As the nodes are connected in a regular random graph with a low goal for the number of links, simple flooding is a great strategy for Inner-Cluster Dissemination. In Streamr, nodes propagate each message they receive from any of their connections to rest of the connections. If a message is detected as a duplicate, it will not be propagated. Duplicate detection maintains a sliding window of the last seen 10000 ranges of message sequences. If a received message is within one of the ranges, it is counted a duplicate. Once the length of held ranges is above 10000 the last range is dropped from the map. When gaps are not detected the algorithm will only hold a single range between the first and previously received unique message. Simple flooding is used for Inner-Cluster Dissemination instead of propagating the events to a random subset of neighbors as it would go against the *message completeness (6)* and *low and predictable latency (3)* requirements for the Streamr Network. Adding random components to flooding would make message delivery probabilistic in nature which makes it inappropriate for the aforementioned mentioned requirements.

Even though regular random graphs with the node degree of four should always be fully connected [90], there is always a chance for messages to be lost in the real world. To combat this, mechanisms for detecting lost messages have been introduced to the protocol. Each published message contains a sequence number and a reference to the last message in the sequence. These fields can be used by receiving nodes to detected lost messages and to fill in gaps. The gap filling is not automatically performed during flooding, but is provided as a higher level user feature in the broker node and client software. If the messages arrive out-of-order in real-time, gaps are detected by keeping track of the sequence numbers of the received unique messages. The gaps can be filled by the lost messages arriving via another path in real-time or alternatively with resend requests if necessary. To make resends possible, willing nodes can be assigned as storage nodes for streams. If nodes detect lost messages, they can send resend requests to the storage nodes. The storage nodes can be found by querying a storage node registry Smart Contract or by asking the responsible tracker of the stream for the presence of known storage nodes. Upon receiving information of the storage nodes, WebRTC or WebSocket connections can be formed to perform queries for lost ranges of data.

Despite the benefits of simple flooding in random graphs, it does come with some inherit challenges. Increasing the node degree of a stream overlay comes with the benefit of improving the latency, churn tolerance and message completeness of the stream. However,

each added connection will also linearly increase the bandwidth requirements for each node. This means that there the fulfillment of the *message completeness (6)*, *churn tolerance (7)*, *low and predictable latency (3)* and *simplicity (11)* requirements come with accepted cost for the *bandwidth efficiency (5)* requirement.

As the Streamr Network is moving towards decentralization by making running nodes possible for users, the contents of the messages will be encrypted on the application layer. The users' Ethereum keys will be used to make this possible. This will also make it possible for receiving nodes to validate the source of the message.

4.3 WebSocket and WebRTC Implementations

The connection layer of the Streamr Network has been implemented with two protocols: WebSocket and WebRTC. Both implementations are separate versions of the network and are incompatible with each other. The WebSocket version was the first to be deployed into production in the Corea release [70]. Both versions use WebSocket for tracker-to-node connections to allow bidirectional communication. However in the first version, all node-to-node connections use WebSocket and in the second all node-to-node connections are formed with WebRTC. There are considerations to create a hybrid solution where connections formed with both protocols can exist in the same stream overlays. One of the biggest benefits with WebSocket is the ability to not use TLS encryption to save computational costs. In the WebRTC protocol, the use of DTLS is mandatory. However, as described in section 2.2.2 WebRTCs transport layer protocol SCTP is better for the requirement of *optimization for small payloads (4)*.

The WebSocket implementation is the simpler out of the two. It was not built to be used outside of internal data center setups and thus made no attempts at NAT traversal. This made the architecture and protocol of the WebSocket version much simpler. Each node is responsible for running a WebSocket server and informing the tracker of its public IP address. Whenever the tracker instructs two nodes to connect to each other they will receive each other's IP address with the instructions. Both nodes will then attempt to act as a client and form a connection. As a result, two connections should be formed. To avoid duplicates, one of the connections is killed based on a simple condition.

The WebRTC implementation adds additional responsibilities for the trackers. As described in section 2.2.6, signaling servers are required in the WebRTC protocol. They are quite conveniently placed in the trackers of the Streamr Network. Thus, the tracker-based

signaler can be used to easily validate if any requested node-to-node connections have not been instructed by the tracker. This adds a layer of security and *attack resilience* to the system. WebRTC also adds some complexity to the formation of connection and thus the fulfilling of tracker instructions. It takes much longer to form a connection in WebRTC than in WebSocket as ICE is used by default. Based on the experiments in section 5.4, it was found that when STUN is used forming WebRTC Data Channels between high latency pairs can take up to 12 seconds. Even if STUN or TURN are not required, each connection must first be negotiated via the tracker-based signaller. Once the signaling is over, a DTLS connection is negotiated after which an SCTP association is negotiated via the DTLS connection. The mandatory use of DTLS could be unnecessary in the Streamr Network as there are plans to encrypt messages in the application layer. This could affect the throughout and latencies of the network. Moreover, it is possible to enable P2P use of WebSocket in the presence of NATs with the ICE protocol and signaling procedure used in WebRTC.

The tracker-to-node communication is key for the overlay construction in the Streamr Network. This area of the protocol has seen the most improvements and changes outside of the change to use WebRTC instead of WebSocket for transportation. The tracker-to-node communication in question does not include signaling but the flow of instruction and status messages used for overlay construction in the Streamr Network. The differences in the flows are depicted in figure 4.2

In the WebSocket version the tracker-to-node communication was more naive than in the WebRTC version. After each instructed connection was formed, the node would add a status message containing all of its stream states to an outgoing queue of size 1 with a timeout of 200ms. Whenever a new status message would be added to the queue, the timeout would be reset. Once the timeout would expire, the newest status message would be sent to the tracker. Once the tracker received a status message it would check if it conflicted with its currently known stream overlay state. The tracker would update its state based on the status message and generate new instructions to any affected nodes. The nodes would process all received instructions in parallel. Due to this approach, it is possible that when a node was handling multiple instruction at once, it could report its stream state back to the tracker before finishing the handling of instructions. In some cases, the node degree could already be full of connections to low delay nodes while still handling instructions to high delay nodes. This is depicted in figure 4.2 where the node is processing instructions, yet it has sent a status message containing 4 connections

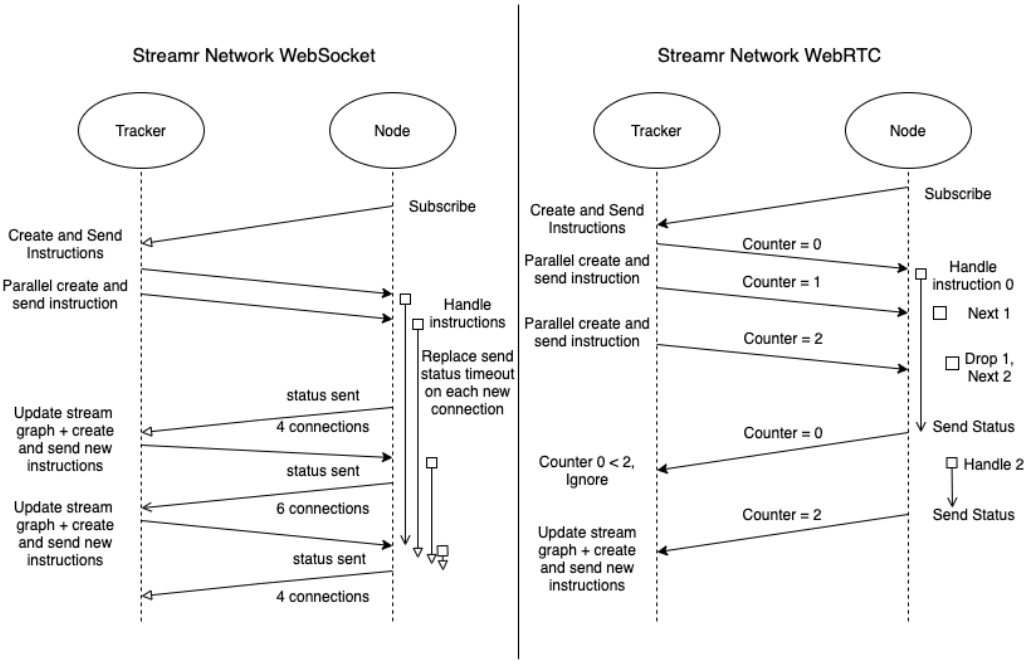

Figure 4.2: The tracker-to-node communication mechanisms of the WebSocket and WebRTC implementations of the Streamr Network

back to the tracker. The tracker could also receive status messages with more stream connections than the node degree allows. In these cases, it would instruct the node to disconnect from any overflowing connections. This approach could have led to accidental underlay topology awareness (TA) in the overlay construction mechanism. In chapter 5, proof for how common such occurrences were is shown. Although, TA is a great feature for improving the performance of the stream overlays, the tracker-to-node communication had several problems in the Streamr Network. First off, it is obvious that the nodes violate the instructions of the tracker and end up with incorrect stream connections. This would often include violations to the node degree of the stream overlay if the nodes joined the stream at a fast pace. Most importantly, the WebSocket implementation was very wasteful in terms of resource consumption as network bandwidth and CPU was wasted. In large stream overlays it could take minutes for the overlay construction to resolve *after* churn.

The WebRTC version has seen significant improvements to *churn tolerance (7)* and control data related *bandwidth efficiency (5)* in the communication between the node and tracker. Counter based throttling was introduced to the flow of instruction and status messages. Each instruction has a counter kept up to date by the tracker and each status message includes the counter of the latest instruction message. The first use for this mechanism is that the trackers can filter out of date status messages. The second, is

that the nodes will only ever process the latest instruction based on the counters. Importantly, nodes only send one status message back to the tracker per handled instruction. These mechanisms significantly improved the performance of the Streamr Network under churn. Moreover, the WebRTC implementation introduced reliability measures for handling tracker instructions. Nodes store the latest received instruction for all streams and reattempt the instructions periodically. If the instruction handling fails, the nodes will send a status message to the tracker to ask for a new instruction.

All in all, the WebRTC implementation of the network is starting to be in a very stable state and has been in production use for some time now. However, there is research to indicate that WebRTC's underlying transport layer protocol SCTP does perform worse on average under normal network conditions when compared to WebSocket's TCP as seen in section 2.2.2. Moreover, DTLS is a mandatory part of WebRTC and cannot be turned off. Turning transport layer security off could be useful in the public Streamr Network where messages will be encrypted in the application layer. As WebSocket does not mandate the use of TLS, switching back over to WebSocket could be necessary for better throughput. Moreover, WebRTC is not unique in its capacity for NAT traversal as it is possible to add the NAT traversal mechanisms used in WebRTC to any UDP based transport protocol. There has even been drafted solutions for using the WebSocket protocol over TURN servers [16] making ICE a real possibility for WebSocket. Despite this, as long as browsers will not be able to run WebSocket servers without significant development efforts, WebRTC will be the best way to make Streamr Network nodes executable in browsers and to form general connections between peers behind NATs.

5 Theoretical analysis, Simulations and Real World Experiments

The design and implementation of the Streamr Network are validated using results of theoretical analysis, simulations, emulations, and real-world experiments. Simulations and theoretical analysis are used show that the Streamr Networks tracker generates d-regular random graphs as its topologies and to demonstrate the implications, tradeoffs, and performance estimations due to this in terms of the stream overlays. The emulations and real-world experiments were ran against both the WebSocket and WebRTC node-to-node overlays to validate the implementations. Finally, the results of the real-world experiments are compared to those of previous related work.

During this chapter, the following metrics are used to measure the performance of the Streamr Network:

- *Node degree* is used to express to number of connections each node has in a stream overlay.
- *Network Diameter* is the longest shortest path between any two nodes in a stream overlay. It is expressed as the number of hops.
- *Round-trip time* (RTT) is used to express the time it takes in milliseconds for a message and its corresponding acknowledgement in the underlay to be exchanged between two nodes. This makes it double the time of a one-way delay between the same nodes.
- *Relative delay penalty* [18] (RDP) expresses the ratio between a one-way delay of the times it takes for a unicast message to be sent and received between two nodes in the underlay and overlay. RDP is sometimes referred to as Latency Stretch [68, 92] in the literature.
- *Average relative delay penalty* is the average of all of the RDPs measured in the stream overlays.
- *Message propagation delay* expresses the time it takes in milliseconds for all nodes to receive a message that a node has published.

- *Message redundancy* is the metric for the mean number of times that nodes receive a unique message in a stream overlay.

5.1 Theoretical analysis and simulations

The tracker constructed stream overlays in the Streamr Network are design to reassemble d-regular random graphs. This section shows that the tracker generated graphs do reassemble d-regular random graphs and explores the theoretical implications of this in terms of expected performance.

5.1.1 Tracker simulation

To prove that the tracker generated stream overlays are d-regular random graphs, simulations were performed on the tracker algorithm described in section 4.2.1. To prove d-regularity it is necessary to show that the node degree of all nodes in the overlay graph is exactly *d*. However, if the number of nodes *n* in the overlay is less than *d*, the node degree should always be *n - 1*. Proving the randomness of the network is almost necessary through simulation as doing so formally is quite difficult.

To prove the randomness of the stream overlays, the probability equation for one node to connect to another node was calculated. The rules for forming connections in the Streamr Network are as follows: First, there can only exist one connection between a pair of nodes at maximum. Second, a node cannot connect to itself. Based on these two assumptions the following equation was formed:

$$P(\mathrm{v}_i \text{ is connected to } \mathrm{v}_j) = \frac{n*\frac{d}{2}}{\frac{n^2-n}{2}} = \frac{d}{n-1}. \tag{5.1}$$

where d is the node degree, n is the number of nodes and v_i and v_j where $i, j \in [0...n]$, $i \neq j$. In the equation, $\frac{n^2-n}{2}$ corresponds to the total number of possible connections between all nodes in a random network with the two rules for forming connections accounted for. $\frac{d}{n-1}$ accounts for the total number of possible connections in a d-regular graph.

To validate that the graphs generated by the tracker algorithm are d-regular random graphs, the algorithm was extracted from the code base and simulations were performed on it to generate graphs of n = 1000 and d = 4. Based on equation 5.1, the expected

probability for a connection existing in this scenario is $\frac{4}{1000-1} = \frac{4}{999} \approx 0.0040040$. The simulated nodes were given distinct identifiers between the range of 0 and 999 and were injected in the algorithm in order. The simulation was repeated 1000 times after which the total numbers of connections between two distinct pairs was calculated. The connection counts are depicted as a matrix in figure 5.1. In the matrix, each pixel presents a connection between two distinct nodes. The darker pixels represent connections that were present more often, and conversely the lighter pixels represent fewer common connections.

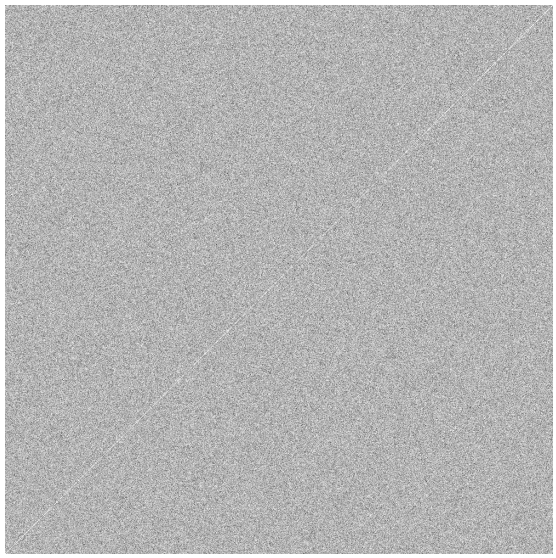

Figure 5.1: Connection matrix between pairs of 1000 simulated nodes with a node degree of 4 after 1000 repeats of the tracker's graph generator algorithm

A white line can be seen passing through the diagonal in figure 5.1. The line represents the fact that nodes could not form connections to themselves. The line can also be seen as a mirror between the two sides in the matrix. All pixels are mirrored by the diagonal because all the connections in the graph between a distinct pair are two-way connections. As the matrix reassembles white noise, the graphs generated by the algorithm are indicated to be random in nature. Moreover, the mean probability for a connection existing between a distinct pair in the simulations was 0.0040040. This number matches exactly with the expected probability calculated with equation 5.1.

Further analysis of the results shows that the distribution of probabilities is spread around the mean. However, there seems to be an up to three times increased probability for a connection to exist between the first few nodes that join the overlay topology. The increase was detected between the nodes with the lowest identifiers as they always joined the overlay first. Despite the slight long tail in the probability distribution, the graphs generated by

the Streamr Network tracker algorithm can be seen to closely reassemble d-regular random graphs.

5.2 Theoretical analysis for expected flooding performance

Based on the proof that the stream overlay graphs of the Streamr Network reassemble d-regular random graphs, the optimal performance and behavior of the overlays can be predicted based on the related literature. Based on related work on d-regular random graphs, it is also quite simple to point out any specific practical tradeoffs for tuning the stream overlays.

To predict the *network diameter* of the Streamr Network the theoretical results presented in [90] can be used. The equation used in the paper for predicting the *network diameter* $diam$(G) for a random graph G, with a node degree d and number of nodes n is as follows:

$$\begin{aligned} 1 + [log_{d-1}\ n] + \left[log_{d-1}\left(\frac{(d-2)}{6d} log\ n\right)\right] \leq \\ diam(G) \leq 1 + [log_{d-1}((2+\epsilon)\ d\ n\ log\ n)]\,. \end{aligned} \tag{5.2}$$

It is necessary to point out that the equation 5.2 presented in [90] does not account for any possible edge weights present in the graph. In the context of P2P networks any latencies between connections in the graph can end up affecting the upper bound of the *network diameter*. Thus, the upper bounds of the network diameter predicted by the equation only apply to P2P networks where latencies between connections are equal. In figure 5.2, the upper bounds for *network diameter* with node counts ranging from 0 to 100 000 and even node degrees from 4 to 16 are depicted based in equation 5.2. By plotting the output of equation 5.2, it is clear to see the network diameter of d-regular random graphs increases logarithmically with the number of nodes in the network.

The metric of *message redundancy* is very useful for evaluating how well the requirement for *bandwidth efficiency (5)* is met. This follows from the fact that redundant messages exhaust the bandwidth of network nodes. As mentioned in section 4.2.2, there is an accepted tradeoff between the requirement for *bandwidth efficiency (5)* and four other requirements in the design of the Streamr Network.

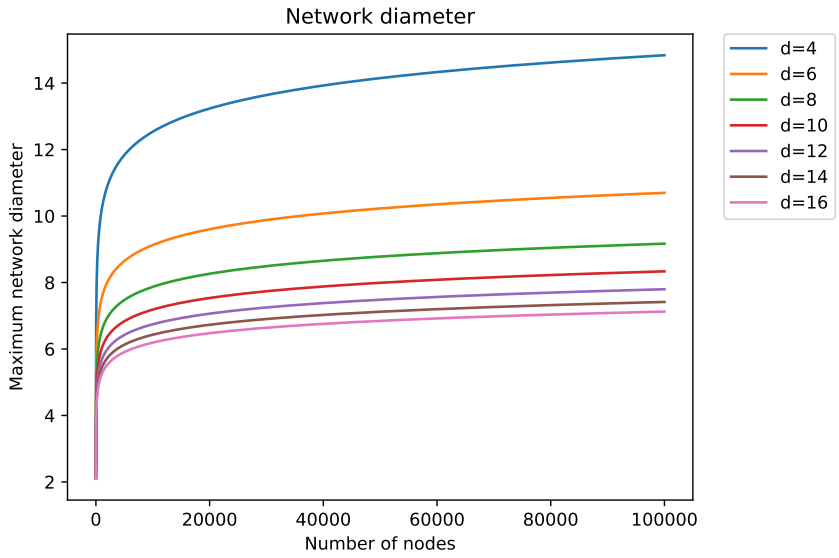

Figure 5.2: Upper bounds of network diameter with varying node degrees and node counts according to equation 5.2.

An equation for the unavoidable upper limit of redundant messages during flooding in random graphs is provided in [49]. However, reaching the unavoidable upper limit comes with the cost of sending handshake messages between neighboring nodes before propagation to decipher if the nodes have already received a given message via another path. This approach goes against the *low and predictable latency (3)* requirement of the network as each hop in the network would cost an extra RTT.

In the Streamr Network, each node sends messages all messages it publishes to each of its d stream overlay connections. However, when a message is propagated it is not sent back to the connection from which it was received. This means that the uplink cost for receiving a message in the Streamr Network is *d-1*. Moreover, it could be possible that in the worst case each node would receive each message d times. In the Streamr Network's Corea Release Whitepaper [70] the expected number of duplicates a non-publishing node was expected to receive was *d-2*. However, experiments performed for this thesis contradict this formula in the presence of latency, as will be seen in section 5.3.1. Based on the experiment results the back-propagation avoidance and duplicate detection mechanisms function better in the presence of latency that in the absence of it.

5.3 Emulation experiments

To compare the performances of the WebRTC and WebSocket implementations under packet loss and to measure the *message redundancy* of the Streamr Network, nodes were ran over underlying networks emulated by the Common Open Research Emulator (CORE) [20] for this thesis. CORE makes it possible to create emulated underlying networks with modifiable bandwidth capacity, jitter, latency, packet loss etc in links between devices. Importantly, only a single Linux machine is needed to run an emulated underlying network. This makes running the emulation experiments much cheaper than the real-world experiments of section 5.4. Moreover, there is a single clock that can be used to accurately measure the ages of published messages with timestamps.

The basic setup with CORE in the emulation experiments was comprised of 16 emulated data centers connected with routers as depicted in figure 5.3. There is a single router in each emulated city and the routers are connected with a minimum of 2 connections to two other routers in the setup. All latencies between routers in the emulated cities were fetched from the Wonder Network [89]. Behind each router of a city is a switch that connects to a given number of emulated host machines. The host machines are placed in the edges of figure 5.3 and have an index number in the tail end of their name. All devices in the emulated underlay network have their own unique IPv4 addresses. This makes NAT traversal methodologies not required during the experiments. All emulated host machines run Streamr Network nodes in the experiments. There is a special emulated host dedicated for running a tracker node in Helsinki.

To find optimal path in the underlying network the Open Shortest Path First protocol (OSPF) [54] was used. OSPF is a protocol used for link-state routing inside individual Autonomous Systems (AS) of the Internet. OSPF makes it possible to split an AS into multiple areas to make routing more efficient. However, in the experiments, all emulated routers, switches, and machines belonged to a single area. OSPF was setup to find the shortest path between the routers by using the latencies of the links as the weights. Given this setup, the measured mean latency when pinging from a node in each city to the rest of nodes in other cities was 98.26 milliseconds.

Once all the emulated devices were started the emulated hosts and routers were instructed to ping through all the routers in the experiment setup. This would confirm that all the emulated devices were started successfully and additionally the construction of routing tables would be sped up. Once the process was completed Streamr Network nodes and

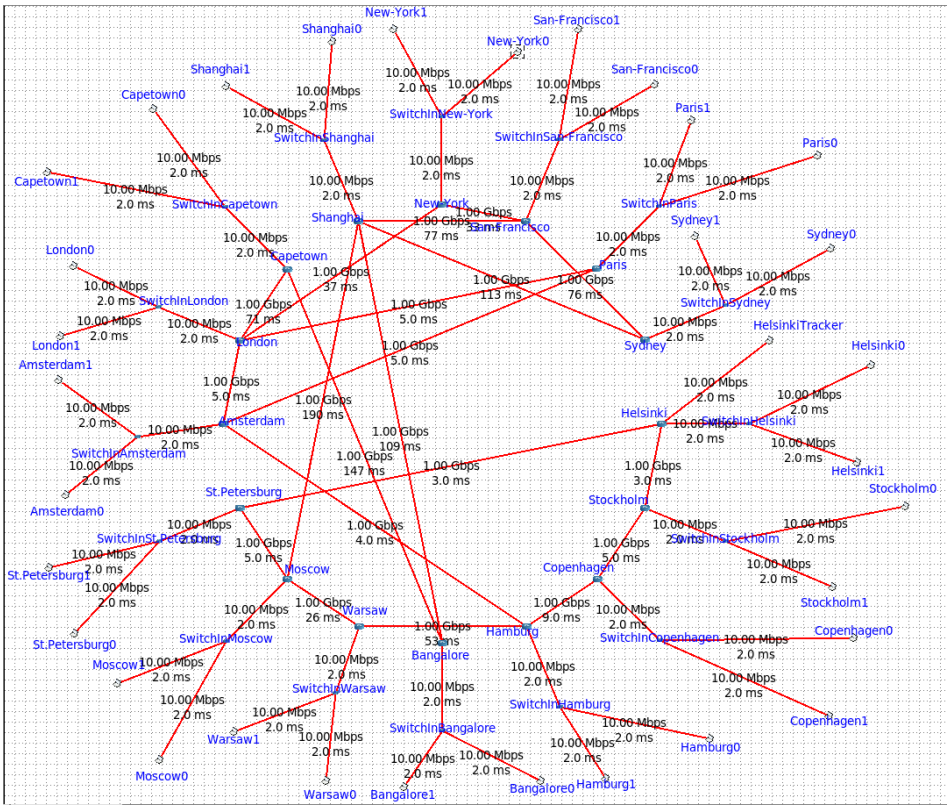

Figure 5.3: Example setup of a CORE emulated setup of an underlying network with two machines in each datacenter.

the tracker could be started on top of the emulated underlay.

Each Streamr Network node in the experiments was equipped with a capacity to log metrics of each received unique message into a file. The metrics contained information of the ages of the messages and the numbers of duplicate messages received per each unique message. The contents of the metrics files could be analyzed to find out the results of the experiments. The metrics contained much more information, but the previously listed two key metrics are the most important ones for the experiments ran for this thesis.

5.3.1 Message redundancy

To measure *message redundancy* under latency in the Streamr Network, CORE was setup to run 4 nodes in each emulated data center totaling to 64 nodes. The nodes were started on each of the emulated machines once the emulated underlying network was fully setup by launch scripts. Once all nodes were started, they would join a single stream with a node degree n and take turns to publish a single message to the network. The experiment

was ran on stream overlays with a node degree of 4, 6, 8, 10, 12, 14 and 16. The results of the experiments' key metric of duplicate messages per unique message is plotted in figure 5.4.

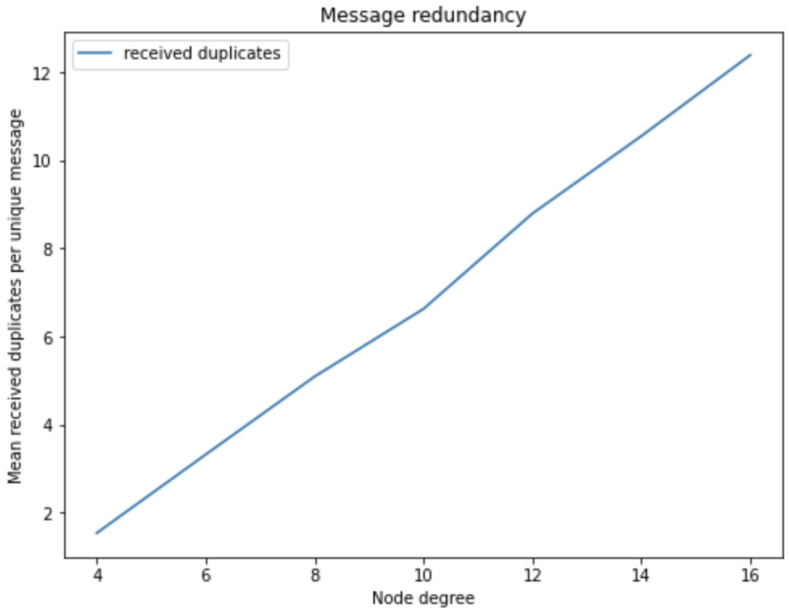

Figure 5.4: Received duplicates per unique message by non-publishing nodes as a function of node degree in emulated networks of 64 nodes.

The results of the experiments show that the number of duplicates received by each non-publishing slightly outperform the predicted performance of *d-2*, where *d* equals node degree described in section 5.2. However, the *message redundancy* remains linear in nature.

Based on the figures 5.4 and 5.2 it is clear to see that there is a tradeoff between the requirement for *bandwidth efficiency (5)* and the requirement for *low and predictable latency (3)*. The *network diameter* directly correlates with the expected *message propagation delay* as the reduction of hops between nodes reduces the expected delay for message propagation. However, by increasing the node degree in a stream overlay the bandwidth consumption of the nodes in the stream increases linearly. Although, in heterogeneous globally distributed real world setups with d-regular random graphs the number of hops in a path does not necessarily mean that its delay is high. It is very much possible that given two paths between two nodes the path with more hops ends up having a lower delay. For example, a path between nodes in Helsinki and Paris with four hops where each intermediate node is situated in Europe can be expected to have a lower latency than a path with two hops where the intermediate node is situated in Sidney.

Moreover, increasing the node degree improves the requirements for *churn tolerance (7)* and *attack resilience (10)*. As more redundant paths are introduced into the network the likelihood of a node receiving a message is increased and eclipsing a node from the rest of the stream overlay becomes more difficult. These two facts make the configuration of the node degree an important task for the managers of streams in the ecosystem. By default, in the current production setup of the Streamr Network and in the real-world experiments performed in section 5.4, the node degree of the stream overlays is 4.

5.3.2 Stream overlays under packet loss

To measure how well the two implementations of the Streamr Network perform under packet loss, CORE was used to emulate networks with packet loss. These measurements are important for *low and predictable latency (3)* and *message completeness (6)* as the quality of service of the network should not be greatly affected under impaired network conditions. Based on the performance comparisons of TCP and SCTP in [48] and [82] focused on bandwidth restricted networks. As WebSocket uses TCP and WebRTC uses SCTP the research can be expected to be reflected in the overlays constructed with the two technologies. However, in the previously mentioned research the performance of TCP and SCTP was not measured against underlay packet loss. As such conditions can be expected in decentralized networks where nodes could be connected to the internet with wireless connections. Even if all connections are wired, some rates of packet loss can be expected in the internet due to network errors or routers dropping packets during congestion.

The underlay topology of the packet loss experiments is similar to the one seen in figure 5.3. The difference being that there is only a single host machine connected to each switch in the cities and that packet loss has been injected to each of the links between the hosts and switches. The bandwidth capacities of the links in the emulated underlay network are similar to the ones in the figure. The packet loss range in the experiments was selected to be between 0 and 5% as the range is defined as random loss in [69].

During the experiments, a Streamr Network node was ran in each of the host machines in the emulated underlay totaling to 16 nodes. The key metric gathered in the experiments was *mean message propagation delay*. Once the initiation of the underlay emulation was finished and the network nodes had subscribed to the default stream of the experiment, each node would publish 250 small messages (total size $<$ 100 bytes) in 50 ms intervals to the default stream. As each node received a unique message, it would add its current

timestamp to the message and store the age of the message into the metric result file of the experiment. The experiments were performed with 0, 0.1, 0.2, 0.5, 1, 2, 3, 4 and 5% packet loss rates and repeated 20 times for each rate of packet loss. The experiments were ran on both the WebSocket and WebRTC implementations of the Streamr Network. The results are plotted as the mean message delivery time for all messages in figure 5.5.

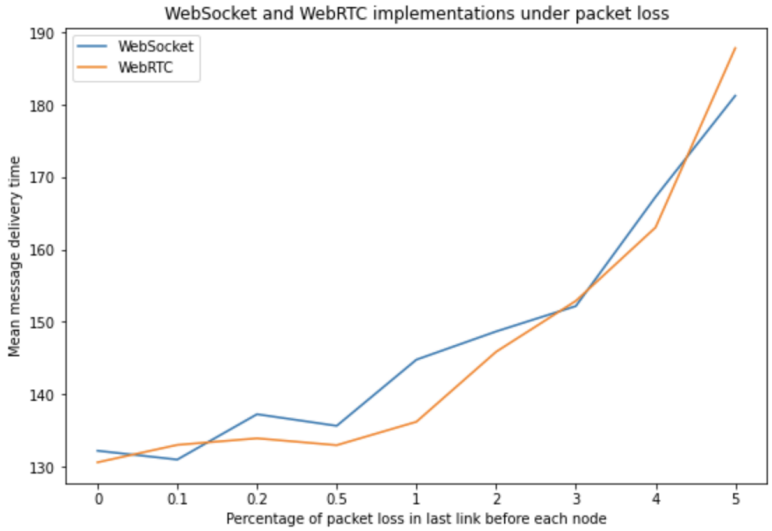

Figure 5.5: Mean message delivery times for the WebSocket and WebRTC implementations of the Streamr Network under packet loss.

Based on the results of the experiments it is clear to see that there is no significant difference between the WebRTC and WebSocket implementations under packet loss. However, it does seem that between the range of 0.2% up to almost 3% packet loss WebRTC stream overlays outperform the WebSocket version. On the other hand, it does seem that WebSocket clearly takes over at 5% packet loss. These observations are explainable by the following factors. SCTP is more efficient in resending small messages if the number of lost packets at once remains low. This is especially the case if the size of the message is significantly smaller that a single MTU as is the case in the experiments. However, if there are more gaps than a SACK message is capable of reporting TCP takes over. This can be seen in figure 5.5 as the packet loss increases over 3%. TCP's fitting of multiple below 1 MTU messages into a single segment should also explain why SCTP is slightly faster under zero packet loss. All messages are immediately cleared from the send buffer in SCTP, whereas in TCP the sender waits for the buffer to fill up or for a small timer to run out as described in sections 2.2.1 and 2.2.2. On top of this, if TCP segments containing multiple messages are lost at low rates, message delivery times will increase as all

messages contained by the segment are lost at once. This is the most probable cause for higher delays in TCP between 0.2 and 3% packet loss.

The experiments indicate that under mostly normal conditions where the bandwidth of a link is not affected but packets are lost at rates below 3%, the WebRTC implementation of the Streamr Network appears to outperform the WebSocket implementation, whereas WebSocket seems to be yield better results as packet loss rates rise above 3%. Moreover, with packet loss rates below 1% the WebRTC stream overlays appear to be more stable and remain closer to the results of the baseline 0% packet loss experiment. This indicates that the WebRTC stream overlays satisfy the requirement for *low and predictable latency (3)* under normal rates of packet loss for the internet. It is worth mentioning that expected the error rate of the performed experiments is quite high. This is due to the tracker generated topologies and how often a message is truly lost being random. To confirm the results, wider experiments with static stream overlay topologies and increased numbers of sent messages should be performed.

5.4 Real-world experiments

The real-world experiments were set up using between 32 and 2048 virtual machines deployed in Amazon Web Services (AWS). The virtual machines were placed evenly across 16 regions in AWS to create a globally distributed network. The used regions and the underlay delay measurements between them are listed in Table 5.1. The goal of the AWS experiments is to validate the design and implementation of the Streamr Network especially in terms of *scalability (1)* and *low and predictable latency (3)*. The two requirements were validated by measuring the key metrics of *message propagation delay* and *relative delay penalty*. On top of the original effort to validate the WebSocket implementation in [70] for which the writer of this thesis was responsible for, the thesis includes similar measurements against the WebRTC implementation of the Streamr Network.

5.4.1 Experiment setup

The nodes in the real-world experiments were deployed in AWS in Elastic Compute Cloud (EC2) [2] virtual machines (VM). The instance type used for running the nodes was "t3.small". In a given region all EC2 machines were scaled up and down with the use of Auto Scaling Groups (ASG) [3]. ASG is a service in AWS used to set minimum, maxi-

mum and goal target numbers of EC2 instances with a given configuration. In the AWS experiments a single ASG was deployed in each of the 16 regions. The ASGs controlled the number of EC2 instances that were started in the experiments. This was important as only one node was run in each EC2 instance.

To initiate and tear down the experiments, the Boto3 library [10] built for controlling AWS with Python3 was used. Boto3 made it possible to create scripts for controlling the experiments with a central point. To install and start the Streamr Network nodes, preinstalled Unix scripts were executed on each EC2 instance upon being started. Systemd scripts were used to start the node processes and to ensure that the processes were restarted after failures or commanded kills. Each EC2 instance was assigned a unique global IPv4 address to ease access to the nodes. However, the IPv4 addresses were assigned to the nodes with the use of full cone NATs by AWS. This made STUN servers necessary for forming WebRTC PeerConnections as the instances were not aware of their own public IP by default. In the WebSocket experiments the IP address of the EC2 instance was passed as a parameter to the node process in the unix scripts.

HTTP endpoints were added to each Streamr Network node to make controlling them possible during the experiments. The endpoints were used to command the nodes to ping a set of IP addresses of other EC2 instances running nodes, to join streams, to publish messages to streams and to kill the node process. Endpoints for querying information from the nodes were also added. An endpoint for fetching information of a node, its status and the key metric data gathered during the experiments were added. The nodes were extended to store key metrics of the ages, hop counts, and received duplicates of each received message. To ensure that the measured message ages were minimally affected by clocks running out of sync, Chrony [38] AWS's improved version of the Network Time Protocol (NTP) [53] was used to maintain a global time during the experiments. NTP is a UDP based protocol that is used to synchronize the clocks of multiple machines by each machine distributing their exact time data. It is designed to take any latency and jitter between the machines into account.

The one-way delay data seen in Table 5.1 was measured by deploying 64 nodes in each of the 16 regions. Each node was commanded to ping the IP addresses of the other 1023 nodes in the experiment. The measured RTT values were divided by two to produce mean one-way delay measurements between all the regions. The RTT data is used in the thesis for calculating the RDP values for both the WebSocket and WebRTC experiments in section 5.4.5 and to estimate optimal performance for the stream overlays in 5.4.4.

Table 5.1: Mean one-way delays between AWS deployed nodes in different regions is milliseconds

	euc1	euw1	euw2	euw3	eun1	use1	use2	usw1	usw2	cac1	aps1	apne2	apse1	apse2	apne1	sae1
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

An additional EC2 instance was also required for running a tracker node. The EC2 instance type used for the tracker was "c5.large". The tracker instance was setup with ASGs, Unix scripting and Systemd. The tracker node was extended with HTTP endpoints that were used for fetching the stream overlay states from the tracker point of view and killing the tracker process if restarts were required. The tracker instance had to be manually started and a domain name used for accessing the tracker by the driver script and the network nodes had to be mapped to the ip address of the instance of the tracker. This made the tracker EC2 instance a static point during the experiments in and the same instance was often used during separate experiment run times.

5.4.2 Experiment runtime

The AWS deployed real world experiments were controlled with pre-programmed Python3 driver scripts during run time. The experiments were split into 4 distinct phases. The phase 1 of the experiments was started by checking that a tracker was accessible behind the specified domain name with the use of HTTP requests. If the tracker was found an HTTP request to kill the tracker process was sent. The script would then wait for the tracker to restart after which it would continue with the experiment.

In phase 2, Boto3 was first used to launch a set number of EC2 instances per region with the use of ASGs. The goal numbers per region varied from 2, 4, 8, 16, 32, 64 and 128 instances. A node was started in each of the started instances totaling to 32, 64, 128, 256, 512, 1028 and 2048 Streamr Network nodes. Boto3 provided key information of the launched instances which was used to conduct the experiments forward. The IP addresses of each instance were accessed and mapped to the driver script stored region-based names

of the instances. Once the driver script had received all IP addresses it would confirm that all of the instances were started successfully based on instance states provided by Boto3. After instances were successfully launched the next phase could begin.

Phase 3 was started by polling all the HTTP endpoints of the nodes based on IP addresses. This was done until all endpoints would respond successfully. Once all nodes were confirmed to be started, they were instructed to subscribe to the default stream used in the experiments. Once the subscription phase was completed, the nodes were instructed to publish messages to the default stream in series. Two nodes at a time were instructed to publish 2 below 100 byte sized messages in 3 second intervals. The next instructions for publishing were sent after the driver received responses from the previous 2 nodes informing that publishing was started. This resulted in an average of 13.6 messages being published to the default stream per second. Once the message propagation in the network was completed the driver script sent HTTP requests to download key metric data from all of the nodes. In phase 3 the final action was to kill all of the nodes for restarts.

In the WebSocket implementations experiments, the subscription step of phase 3 had to be throttled for stream overlays of size 1000 nodes and higher due to the design fault in the tracker-node communication protocol described in 4.3. Without throttling, the stream overlays could take minutes to fully resolve after the last node joined the overlay and were not d-regular upon completion. In the experiments performed for the WebRTC version, throttling was not required and the nodes could join the streams at very a high pace without affecting the outcome of the overlay construction. This shows that the change to the tracker-node communication protocol had significant improvements to the systems *churn tolerance (7)*.

In phase 4 of the experiments were torn down. In this last phase, all EC2 instances used in the experiment were shut down by setting the desired count for ASGs in all regions to zero with Boto3. If there were no further experiment phases to repeat, the driver script would exit its process and data analysis could begin.

By splitting the experiments into distinct phases, the run time became much simpler to handle. The phase 3 of the experiments could be used to perform completely different types of experiments on the Streamr Network. For example, the delay measurements of table 5.1 and the mean propagation delay experiments were performed with separate phase 3 scripts. Moreover, running n repeat experiments for a specific node count was quite simple, as the same EC2 instances could be used between repeats by repeating phase 3 n times. If the number of instances had to be increased for larger node count experiments

the driver would simply enter phase 2 to start up a new set of EC2 instances to launch new network nodes on. In this thesis, 10 repeats were performed for the specific node counts between 32 and 2048 for both the WebRTC and WebSocket implementations of the Streamr Network.

5.4.3 Message propagation delay

The main purpose of the real-world experiments was to measure how the key metric of *message propagation delay* is affected as the number of subscribing nodes is scaled up in a globally distributed setting. To get proper metrics, publishing nodes would include a modifiable *age* field in milliseconds and a *last seen* timestamp to each message to measure the amount of time it takes for each node to receive the message. Whenever a non-publishing node received a unique message, it would subtract its current timestamp from the last seen timestamp and add the result to the age field. Each node would also store the summed age to the key metric data it gathered. As described previously in the chapter, the key metrics data was fetched from all the nodes after each phase 3 of the experiment run time and the clocks were kept in sync with Chrony during the experiments.

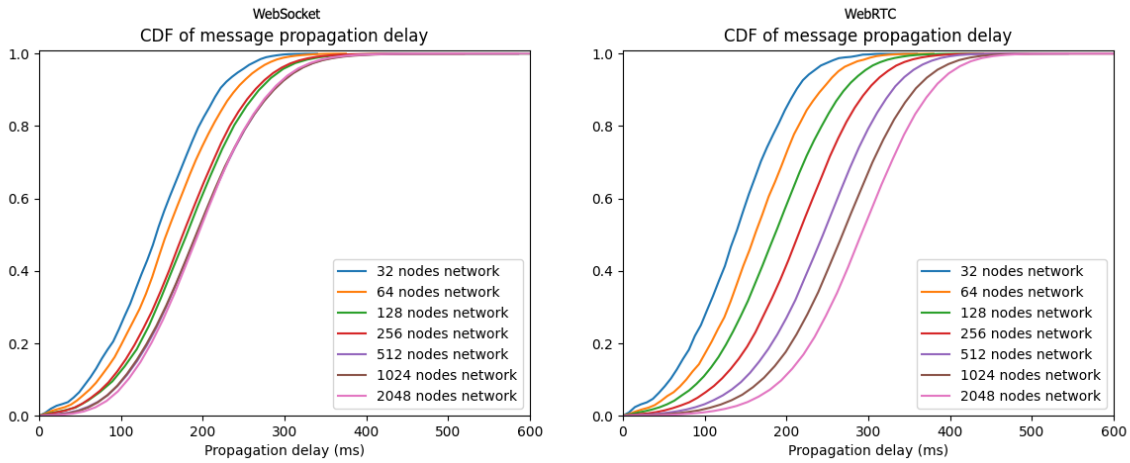

Figure 5.6: Cumulative Distribution Function of message propagation times in the WebSocket and WebRTC implementations of the Streamr Network.

From figure 5.6, it is clear to see that the WebSocket implementation of the Streamr Network outperforms its WebRTC counterpart as the number of subscribers in the network is scaled up. To validate the stream overlays of the WebRTC and WebSocket experiments, metrics of node-to-node connection counts and total numbers of received messages were gathered. Based on the gathered validation data it is clear that all subscribers received

all published messages during the experiments. This indicates that the requirement for *message completeness (6)* is fulfilled. All nodes apart from very few cases formed exactly 4 connections. This indicates that the increase in the message propagation delays in WebRTC must be due to each hop being more costly or the stream overlays being less optimal. As described in 4.3 it is possible that the WebSocket implementation could unintentionally prioritize the formation of low delay connections in the stream overlays due to a design flaw its tracker-to-node communication protocol.

Figure 5.6 shows that in the WebSocket implementation, 99% of all messages were delivered within 362 milliseconds even when 2048 subscribers had joined the network in a global setting. For WebRTC the 99th percentile for message delivery was at 444 ms. At first glance it seems that each hop in the WebRTC setting adds significant delay during message propagation. To verify this, I specifically looked through messages propagated via connections with extremely low measured one-way delays. Based on this data it appears that there were no noticeable differences between the ages of hops in and the measured underlay delays. This indicates that the cause for the increase in *mean message propagation delays* of the WebRTC stream overlays is not due to the transport protocol.

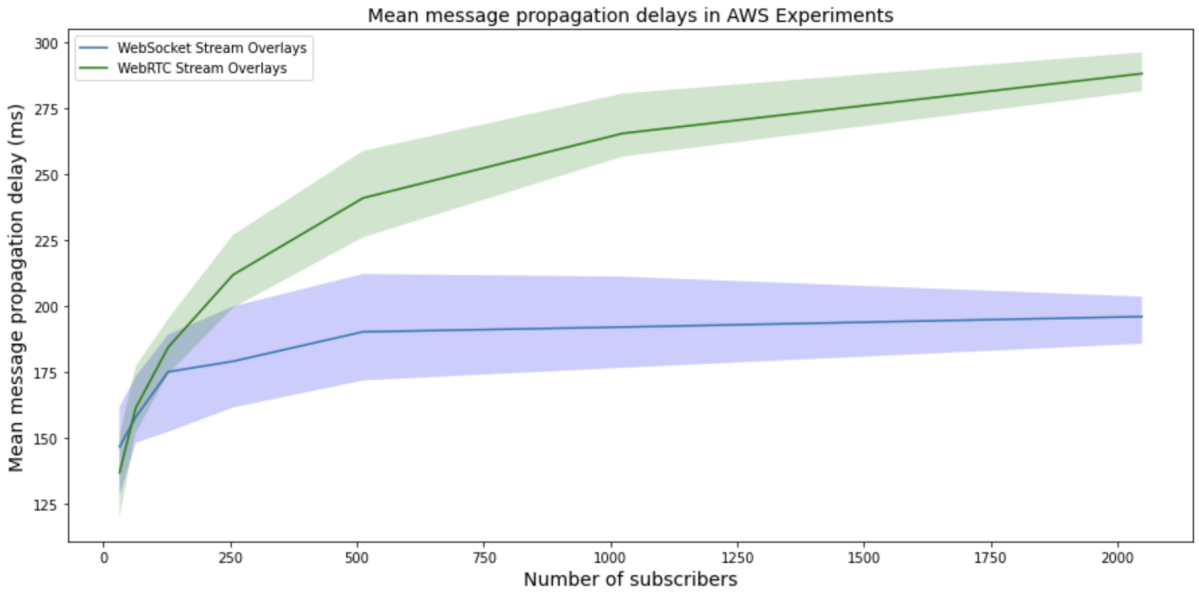

Figure 5.7: Mean message propagation delays in the WebSocket and WebRTC implementations of the Streamr Network.

For a clearer view of the scalability of the two implementations, the mean message propagation delays of the experiments are plotted in figure 5.7. The dark line in the plots represents the mean message propagation delay of the experiments throughout all re-

Table 5.2: Average message propagation delays in the AWS experiments

	WS	WS	WS	WS	WS	WRTC	WRTC	WRTC	WRTC	WRTC	BOTH
Number of nodes	Min of average delays (ms)	Avg of average delays (ms)	Max of average delays (ms)	Min-Max difference (ms)	Difference%	Min of average delays (ms)	Avg of average delays (ms)	Max of average delays (ms)	Min-Max difference (ms)	Difference%	Difference in avg of avg delays (ms)
32	127	146	160	33	26.0	120	137	150	30	25	-9
64	147	157	172	25	17.1	152	161	177	25	16	+14
128	151	172	186	35	22.9	174	184	195	21	12	+22
256	160	177	197	37	23.1	199	211	227	28	14	+34
512	170	189	211	40	23.7	226	241	259	33	15	+52
1024	175	190	210	35	19.8	256	266	280	24	9	+76
2048	184	194	202	18	9.8	282	288	296	14	5	+94

peats. The shadows in the figure represents the range between the max and min mean message propagation delays of any repeat during the experiments. The shadows were plotted as each repeat of the experiments used a freshly generated stream overlay topology. The figure shows more clearly that the WebSocket implementation scales much better than its WebRTC counterpart. Despite the WebSocket implementation outperforming its WebRTC counterpart, it is clear to see that both still exist within logarithmic scale. The parameters of the graph can be found in table 5.2. The table further analyses the variance present in the experiments. The difference percentage is the main tool used for this and is calculated by dividing the min-max difference by the min of average delays. The comparison between average of average delays between the solutions is calculated by subtracting the average of WebRTC from the average of WebSocket.

The mean number of hops in the WebRTC experiments seems to increase logarithmically as more nodes are added to the network. This is expected based on equation 5.2 for d-regular random graphs. Based on this information it is likely that if the experiments were scaled up to 10000 nodes, the mean message propagation delay for the WebRTC experiments should flatten. In the experiments performed for the WebSocket version the mean number of hops during the experiments was not recorded. However, data for the maximum number of seen hops can be found. The max numbers of hops in the experiments appear to increase near-linearly from 9 to 42 for WebSocket and from 9 to 33 for WebRTC. Long paths are expected due to the experiment setup where nodes only exists within 16 regions of AWS. As connections inside regions have delays of approximately 0.1 ms, the setup is bound to create paths where hop counts are high while the ages of the messages are low. In WebRTC the mean message propagation for paths where the hop counts 25 or over was 334 ms, and for WebSocket the measurement was 224 ms. This shows that the high hop paths had only a slightly higher delay that the mean message propagation of all paths in the experiments. It also indicates that the longer paths in WebSocket appear

to be more efficient as the difference to the mean is 30 ms and 46 ms in WebRTC. This result is quite interesting as the maximum paths are longer in WebSocket which points to bias towards the formation of low delay connections in the stream overlays.

5.4.4 Predictability of latency

To measure how predictable the *message propagation delays* of the Streamr Network are based on underlay delays, Dijkstra's algorithm [65] was used to calculate optimal *mean message propagation delays* in the stream overlays. Dijkstra's algorithm is used to find the shortest path between two nodes in a graph based on distances between all nodes in the graph. It can be easily modified to find the shortest path tree to all nodes from one node in the graph. The shortest path tree was calculated for all nodes in the graphs to find optimal *message propagation delays*. The algorithm was ran over the tracker generated stream overlays in the WebSocket and WebRTC experiments. The graphs were fetched from the HTTP endpoint of the tracker after each experiment repeat. The measured latencies in table 5.1 were inserted as distances between nodes in the stream overlay graphs. The measured *mean message propagation delays* of the experiments and performance estimates of the stream overlay graphs are plotted in figure 5.8.

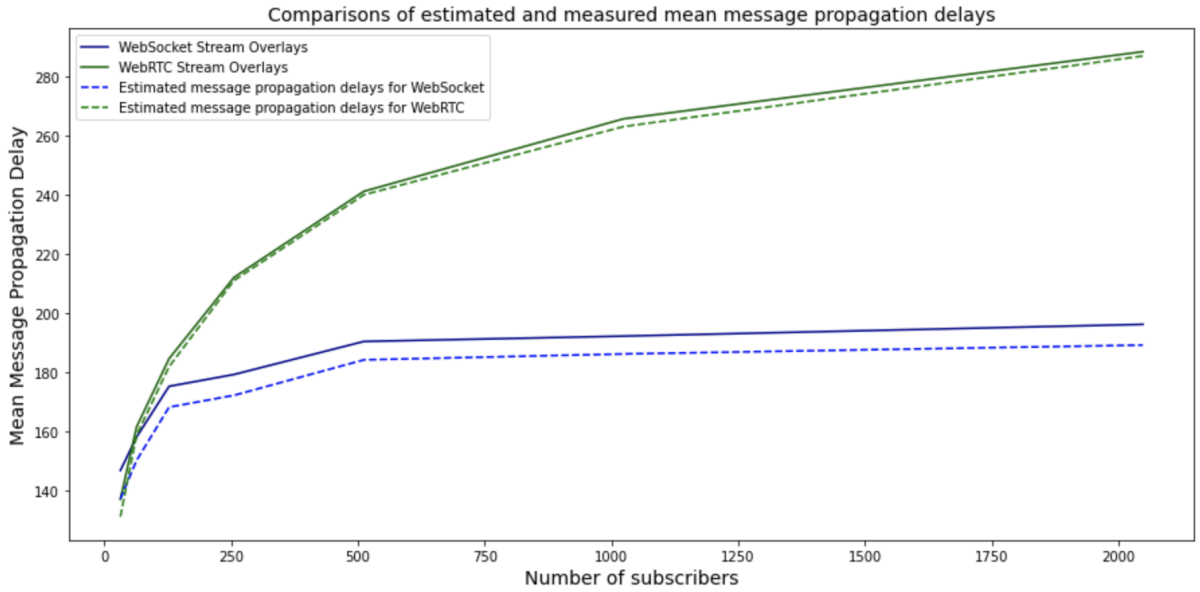

Figure 5.8: Estimated and measured mean message propagation delays in the AWS experiments.

By comparing the estimated and measured delays it is possible to state that the message propagation delays of the Streamr Network can be predicted. It appears that each hop

Table 5.3: Estimation accuracy

	WebSocket	WebSocket	WebSocket	WebSocket	WebRTC	WebRTC	WebRTC	WebRTC
Number of nodes	Average of estimated average delays (ms)	Average of measured average delays (ms)	Absolute difference (ms)	Difference %	Average of estimated average delays (ms)	Average of measured average delays (ms)	Absolute difference (ms)	Difference %
32	137	146	9	6.4	131	137	6	4.5
64	150	157	6	4.6	159	161	2	1.3
128	168	172	4	2.4	182	184	2	1.1
256	172	177	5	2.9	211	211	0	0
512	184	189	5	2.7	240	241	1	0.4
1024	186	190	4	2.1	263	266	3	1.1
2048	189	194	5	2.6	287	288	1	0.3

is less costly in the WebRTC implementation despite transport layer encryption being disabled in the WebSocket stream overlays. The probable cause for the worse predictability in the WebSocket stream overlays is that TCP may package multiple below 1 MTU messages into a single segment. This was also the explanation for the differences in the *mean message propagation delays* of section 5.3.2.

Based on the estimates it is possible to see that the cause for the better performance of the WebSocket implementation is clearly due to overlay construction. If the dashed lines in figure 5.8 would be closer or on top of each other, it would indicate that the graphs are of a similar type. As all topologies had 4 connections based on the trackers view it can be said that the topologies were d-regular. This leaves the difference in the randomness of the graphs as the last probable cause for the performance gap between the two versions. The performance of the WebRTC implementation more closely reflects that of d-regular random graphs seen in [68] and equation 5.2. This leaves bias towards the formation of low delay connections as the remaining cause for the better performance of the WebSocket implementation. As the difference between measured and estimated delays in WebRTC are very low, it can be said that the increase in *mean message propagation delays* is not due to the overhead of the WebRTC protocol.

Based on the results, it is possible to estimate that *mean message propagation delays* of a stream can be estimated with a mean error percentage accuracy (MAPE) of 3.95% for the WebSocket version and 1.24% for the WebRTC version. This means that the delays measured between node-to-node connections could be reported back to the tracker where they could be used to estimate *mean message propagation delays* of streams with low error rates. It is worth noting that node-to-node RTT measurements using the WebRTC or WebSocket protocols instead of ICMP pings would yield more accurate results. By using the transport protocols to measure RTTs, any overhead associated with them would added

to the measurements. Despite ICMP pings being used, the estimations are surprisingly accurate especially in the case of WebRTC where DTLS encryption was used. Furthermore, the MAPE for predictability of latency for WebRTC stream overlays is 0.7% if it is calculated with node counts ranging from 64 to 2048. All in all, the results of this section validate the WebRTC protocol as the better solution *low and predictable latency (3)* and *optimization for small payloads (4)*. Moreover, the lower *mean message propagation delay* of the WebSocket implementation is confirmed to be due to topology construction and not due to a performance gap between the two transport protocol.

Currently in the Streamr Network, nodes collect and report the RTT measurements of their connections back to the tracker. This data is used in the upcoming Network Explorer tool for the Streamr Network which allows visualization of the stream overlays and their real-time metrics. In the future, the node-to-node RTT measurements could be used to improve the stream overlays by making them intentionally underlay topology aware. This would be done by prioritizing low latency connections above random connections during overlay formation. For example, each node in a stream overlay of node degree 4 could be assigned to connect to 2 known low RTT nodes and 2 random connections as in [68]. To ensure that RTT measurements exist for many pairs nodes, random pairs could be instructed to create a connection to measure RTTs and to report the results back to the tracker. This would improve the TA of the stream overlays over time. TA will be further discussed in section 5.5.

5.4.5 Relative Delay Penalty

Comparing the results of different decentralized overlays is often difficult as the used experimental setups are very different. A commonly used metric to alleviate this challenge is *Relative Delay Penalty* (RDP) [94]. RDP is measured by dividing the *mean message propagation delay* of the overlay by the mean end-to-end underlay unicast delays between all peers in the overlay. The mean one-way delay of the measurements was 68.64 ms based on table 5.1. The RDP measurements of the AWS experiments are plotted in figure 5.9.

Based on the RDP results it is possible to understand some of the tradeoffs of the Streamr Network. In the WebSocket overlays, a published message to a stream can be expected to be received by all subscribers within 2.8 times the measured underlay delay between the publisher and any receiving node on average. As the plotted line of the RDP in the WebSocket implementation seems to have flattened by 2048 nodes, the RDP is not

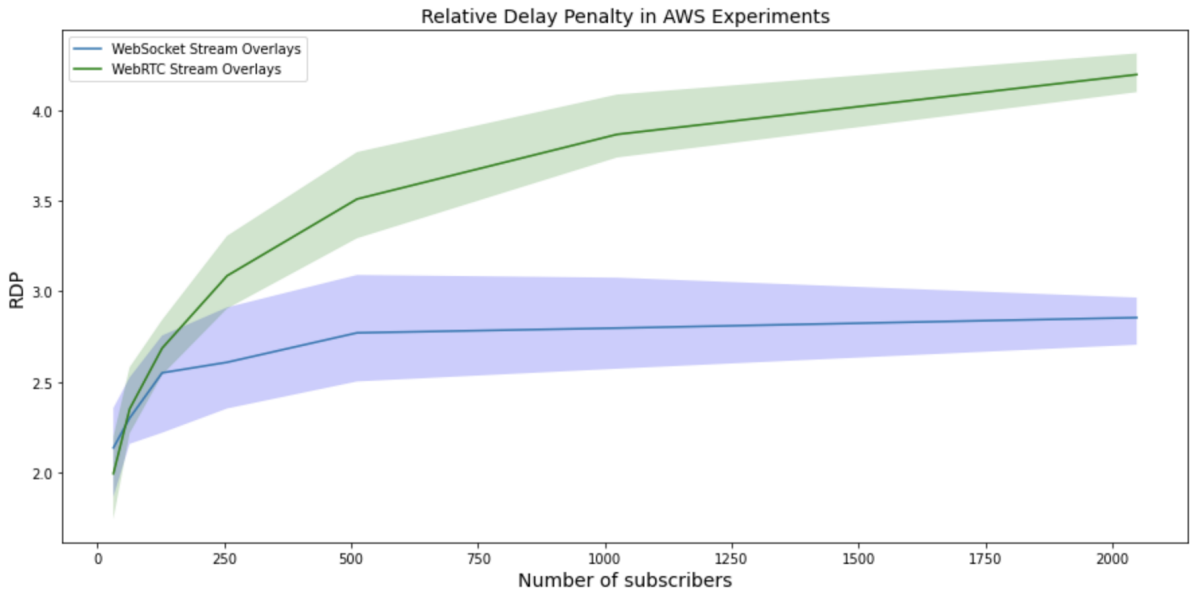

Figure 5.9: Relative delay path penalties of the WebRTC and WebSocket Stream overlays of the Streamr Network.

expected to rise much over the value of 2.8. The RDP for WebRTC stream overlays at 2048 nodes is measured at 4.19. As in figure 5.7, WebRTC's line is expected to flatten by 6000 nodes at the latest. Based on the results, the tradeoff between bandwidth consumption and message propagation delays can be estimated. In stream overlays of node degree 4 with 2048 subscribers, a publishing node initially sends a message to 4 neighbors with the cost of a mean increase of 2.8 times in delays in the WebSocket stream overlays and 4.19 in the WebRTC stream overlays when compared to unicasting the message to all subscribers.

The benefit for *bandwidth efficiency (5)* can be seen as a message has to be sent to 2044 less nodes at once when compared to unicasting. Moreover, as the RDP values remain relatively low, the cost of *low and predictable latency (3)* stays low. As discussed in the previous section, the increased delays of the WebRTC implementation are not due to the transport protocol. This means that with the introduction of WebRTC the overall scalability of the system is improved as nodes can be started behind NATs and in browsers. This feature allows the network to scale in a decentralized manner without the need for network nodes to have interfaces to broker data in and out of clients. This makes WebRTC better in terms of the requirements for *scalability (1)* and *decentralization (2)* despite the tracker having more responsibilities due to signaling.

5.5 Quantitative comparison to related work and topology awareness

As mentioned in section 5.4.5, the gathered RDP measurements can be used to compare the Streamr Network to related work. To do this, similar measurements performed on P2P networks were found from literature. The results of the measurements of this thesis and of related work can be found in table 5.4. The table includes measurements on related work on the structured networks seen in sections 2.3.1 and 3.3.2. On top of the structured networks, RDPs of unstructured networks with overlay topologies based on d-regular random graphs and some tree-based overlays are presented. Some of the solutions are aware of their underlying network topology. This means that they measure underlying latencies or other underlying metrics to prioritize overlay connections. Such overlays are marked as (TA) in the table. The Streamr Networks WebSocket implementation is marker as (TA?). This was done due to the topology awareness of the implementation being unintentional. Moreover, the unintentional topology awareness could be less prevalent in underlays with less variance in delay. This was observed during the emulation experiments with 16 nodes with 0 packet loss in section 5.3.2 where the results of the stream overlays were similar and in the RDP measurements where WebRTC streams outperform WebSocket streams at 32 nodes.

The underlay topologies of all related works listed in table 5.3 were simulated in their experiments. Many of the listed solutions were simulated on several types of underlays. However, to make the results more comparable the transit-stub type underlay was selected if possible in this thesis for simpler comparison. Transit-stub [91] is a model for generating internet type graphs and was originally built to replace the random graph model for simulating networks. It is based on the concepts of transits and stubs. Transits are larger hubs that can host nodes and connect to other transits and stubs. Stubs are smaller hubs that host nodes and can form connections to other stubs but at a much lesser rate than transits. The user can select an edge count that is used to connect the subgraphs formed by hosts in transits and stubs to each other. Tuning the edge count affects the expected delays and connectivity in the simulated underlay.

Instantaneous Reliability Oriented Protocol (IRP) [81] is a non-topology aware tree-based overlay. It was developed to address many of the reliability and robustness issues related to tree based P2P systems. Based on simulations, IRP's RDP was shown to increase sub-linearly from 3.8 to 4.2 in experiments of 700 to 2000 nodes. This makes the results

Table 5.4: Relative delay penalty (RDP) of P2P systems in literature

Ref.	Overlay	Type	Underlay topology	Mean RDP at 1000 nodes	Mean RDP in experiments	Number of nodes in experiments	RDP complexity (N = number of nodes)
	Streamr Network WebRTC	d-regular random graph	Internet	3.8	2.0 - 4.2	32 - 2048	O(log N)
[70]	Streamr Network WebSocket	d-regular random graph (TA?)	Internet	2.8	2.1 - 2.8	32 - 2048	O(log N)
[81]	IRP [81]	tree	transit-stub	3.8	3.8 - 4.2	700 - 2000	O(log N)
[41]	TAG [41]	tree (TA)	transit-stub	1.5	<2	60 - 5000	O(log N)
[92]	Chord [76]	DHT	random graph	5.2-5.7	2.5 - 14	100 - 6500	O(log N)
[92]	LPRS-Chord [92]	DHT (TA)	random graph	3.7	2.5 - 4.5	100 - 6500	O(log N)
[29]	Tapestry [4]	DHT	transit-stub	3	1 - 3	1 - 1020	O(N)
[43]	Pastry [67]	DHT	transit-stub	4.5	2 - 4.5	150 - 1020	O(N)
[43]	Pastry-FNS [15]	DHT (TA)	transit-stub	2.8	2 - 2.8	150 - 120	O(log N)
[43]	GEO-LPM [43]	DHT (TA)	transit-stub	1.8	1.5 - 1.8	150 - 1020	O(log N)
[68]	CAN [60]	DHT	transit-stub	14	10-52	512 - 16000	O(log N)
[68]	TA CAN [68]	DHT (TA)	transit-stub	7	6-14	512 - 16000	O(log N)
[68]	Unspecified [68]	d-regular random graph	transit-stub	3.7	2.5-4.6	100 - 6400	O(log N)
[68]	Unspecified [68]	d-regular (TA) graph	transit-stub	2.8	2.1-3	100 - 6400	O(log N)

remarkably similar to those of the Streamr Network's WebRTC implementation. Topology Aware Grouping (TAG) [41] is a decentralized protocol for forming TA trees. It has the best performance out of the listed systems in table 5.4. Its RDP stays below 2 up to experiments of 5000 nodes. This indicates that TA trees could be the optimal solution to satisfy the requirement for *low and predictable latency (3)*. However, tree-based systems are bound to limit the systems *churn tolerance (7)* as described in 3.2. In the future, it is possible for the Streamr Network to create diverse types of overlays to satisfy requirements of any given stream. For example, if nodes are known to not leave or join the stream often a TA tree-based stream overlay could be optimal.

The RDP measurements for the structured overlays Chord, Tapestry, Pastry and CAN described in section 2.3.1 are listed in table 5.4. The simulation experiments for Chord in [92] are the only experiments performed on top of a random graph underlay topology in the table. It appears that Chord scales logarithmically as a function of the number of nodes. However, its RDP is between 5.2 and 5.7 at 1000 nodes and goes up to 14 at 6500 nodes. This indicates that the Streamr Networks stream overlays significantly outperform Chord. Tapestry and Pastry appear to scale linearly in simulation experiments performed in [29, 43]. However, the experiments are only scaled up to slightly above 1000 nodes. Based on the systems properties, the results should appear to be logarithmic if the experiments would be scaled up. Surprisingly, Tapestry outperforms Pastry quite significantly despite the similarities in the designs of the systems. CAN appears to have the worst performance of all the overlay systems listed in table 5.4. It's RDP is already 14 at 1000 nodes. However, the simulations are performed on 2-dimensional CAN and its RDP performance is bound

to improve by increasing the number of dimensions.

To improve the RDP performance of the structured overlays networks, TA systems have been proposed for them. LPRS-Chord [92] is a TA DHT based on Chord. Basically LPRS-Chord improves the construction by adding a ping phase to the look ups in the system. This requires the use of IP addresses in the application layer of the system. During look ups, each node measures the latency to the next hop and adds to the result. The gathered samples are used to improve the finger tables of Chord in terms of latency. Based on simulation results, LPRS-Chord significantly outperforms its traditional counterpart in RDP. Its performance seems to be on par with the WebRTC implementation of the Streamr Network. A TA version of CAN is presented in [68]. 2-dimensional TA CAN significantly outperforms its traditional counterpart, but its RDP is still measured to be 7 at 1000 nodes.

Pastry-FNS [43] improves its traditional version by adding proximity detection to its neighbor selection and routing. It is shown to perform logarithmically between 150 and 1020 nodes in simulation experiments with RDP results ranging from 2 to 2.8. A new structured overlay designed for TA called Geo-LPM is also presented in [43]. Geo-LPM uses Pastry-FNS for application layer routing but implements a lower-level mechanism to cluster peers that are physically close to each other. This is done with latency measurements and by performing Longest Prefix Matching (LPM) for IP addresses to find nodes that are expected to be in the same physical networks. In simulation experiments, Geo-LPM is shown to significantly outperform the rest of the structured overlay networks. However, the experiments were only scaled up to 1020 nodes where the RDP was measured at 1.8.

In [68] RDP results of different implementations of d-regular graph based unstructured networks were presented. Although, the node degree d is not specified in the paper the results of the experiments very closely reflect those of the Streamr Network. This indicates that experiments were performed with a node degree of 4. The graphs in the experiments were constructed completely randomly or with different types of TA. The best performing TA implementation is listed in table 5.4 as the d-regular (TA) graph. The results are interesting as they are almost exactly on par with the WebRTC and WebSocket implementations of the Streamr Network. Based on the matching results and designs, the RDP of the WebRTC stream overlays can be expected to closely reflect that of the d-regular random graph based unstructured overlay in [68] as more nodes join the streams. This means that at 6400 nodes the RDP of the Streamr Network's WebRTC implementation can be estimated to be approximately 4.6.

The performance of the WebSocket stream overlays are almost identical to the d-regular TA graphs presented in [68]. The TA overlay construction algorithm attempts to form d/2 low delay connections and d/2 random connections. Based on their findings such overlays tend to create long paths consisting of many low delay hops. This reflects the findings of this thesis on the WebSocket stream overlays in section 5.4.3, where the maximum path lengths were much longer than in WebRTC overlays. Based on these similarities it can be said that the WebSocket stream overlays are unintentionally TA. The implication of this is that the difference in absolute *mean message propagation delays* seen in figures 5.6 and 5.7 are not caused by the change in underlying transport protocols. Instead, as seen in figure 5.8 the WebRTC protocol is more suitable for the Streamr Networks performance requirements for *low and predictable latency (3)* and *optimization for small payloads (4)*.

Based on the comparison to related work in this section and the experimentation and analysis in this thesis, it is possible to state that the Streamr Network's WebRTC implementation creates d-regular random graph stream overlays. Moreover, the WebSocket stream overlays had a bias toward the formation of low delay connections in certain circumstances. As the broker nodes strictly follow the trackers' instructions in the WebRTC version, adding intentional TA to the stream overlays is a trivial task. The nodes already send RTT measurements of their connections back to the tracker. Moreover, the public IP addresses of the broker nodes are known by the WebSocket servers in the trackers. Assuming that devices with similar IP addresses are physically close to each other, longest prefix matching of IP addresses could be used for TA similarly to Geo-LPM. This would work especially well for nodes behind the same NATs. As the overlay structure in the Streamr Network is bound to create duplicates, the creation of hot spots in d-regular TA overlays is very unlikely due equalized bandwidth consumption. Thus, the approach would not violate the requirements set by *decentralization (2)*.

6 Comparisons, trade offs and future work

This thesis focused on evaluating the requirements for the Streamr Network described in section 3.2 and to compare the performances and implementations of its WebSocket and WebRTC versions. The requirements for *decentralization (2)*, *churn tolerance (7)*, *zero noise (8)*, *Attack resilience (10)* and *simplicity (11)* were addressed in the design of the Streamr Network in chapter 4. Most importantly, the broker nodes in the Streamr Network were designed to be entirely noiseless. This makes them a great option in IoT deployments where devices have restricted capabilities. The requirements for *scalability (1)*, *low and predictable latency (3)* and *bandwidth efficiency (5)* were validated theoretically and experimentally in chapter 5. The WebRTC and WebSocket stream overlays were compared based on the requirements for *scalability (1)*, *low and predictable latency (3)* and *optimization for small payloads (4)*.

The emulation results show that WebRTC based stream overlays appear to outperform their WebSocket counterparts under packet loss rates of 0.2-3%, whereas WebSocket appears to be better once packet loss rates reach 5% in the network. The results indicated that the predictability of latency remained more stable in WebRTC below packet loss rates of 1% making the solution better for realistic network conditions in P2P networks.

The AWS deployed real world experiments were used to further validate that the latency in the network stays low and predictable as the number of nodes is scaled up for both implementations. The results of the experiments showed that the WebSocket implementation's stream overlays performed similarly to d-regular TA graphs. This confirmed suspicions that the WebSocket version's overlay construction favored low delay connections due to a design fault in its tracker-to-node protocol. However, the protocol had significant down sides in terms of *churn tolerance (7)* and *scalability (1)*. If nodes joined streams at a fast pace, it could take minutes for the overlay construction to resolve. The tracker-to-node communication was significantly improved in the WebRTC implementation which fixed the performance issues of the previous version's protocol. This made it possible for the nodes to join the overlays at a fast pace despite WebRTC connections taking longer to form. Moreover, the improved protocol made the nodes follow the tracker instructions

more strictly. Based on experimentation and analysis in this thesis including comparison to related work, it can be said with certainty that the stream overlays in the WebRTC version of the Streamr Network are d-regular random graphs. The WebRTC version's better tracker-to-node protocol comes with the implication that more diverse types of stream overlays can be constructed reliably with modifications to the tracker's overlay construction algorithm.

The *message propagation delay* measurements of the real-world experiments showed that the predictability of latency was better in the WebRTC stream overlays. The MEPA for estimating mean message propagation delays was measured at 3.95% for WebSocket and 1.24% for WebRTC in stream overlays up to 2048 nodes. The results are great for WebRTC as TLS was not used in the WebSocket stream overlays. This shows that WebRTC fulfills the requirements for *low and predictable latency (3)* and *optimization for small payloads (4)* better than the WebSocket protocol. The experiments also showed that the requirement for *message completeness (6)* is fulfilled in the Streamr Network as all subscribers received all published messages during the experiments.

Overall, the WebRTC version of the Streamr Network is validated to be ready for its Brubeck release where users can start running broker nodes. Thus, the Streamr Network is closer to fulfill the requirement for *decentralization (2)* despite the increased responsibilities of the trackers. Moreover, as all subscribers and publishers in the network can be broker nodes instead of thin clients, the *scalability (1)* of the network is significantly improved. This is due to the brokers not needing to maintain large numbers of connections to outside clients. Instead, they maintain a stable number of connections to other nodes according to the tracker's instructions. Moreover, broker nodes are not required to have public IP addresses and are not required to run WebSocket servers which makes running them in browsers and mobile devices much simpler.

As the thesis did not discuss the fulfillment of requirements for *fairness (9)* and *attack resilience (10)* in detail, further research on these subjects is required. Although, the improved tracker-to-node protocol can be said to fulfill the requirements for *fairness (9)* better than its predecessor. Another path for further research is the search for a transport protocol with a faster handshake period than WebRTC for P2P connections. Moreover, WebRTC's SCTP CRC and DTLS overheads could have unwanted effects to the throughput of the system especially after application layer encryption is added to the Streamr Network. Thus, more efficient datagram-based protocols should be explored. A good starting point for a more efficient overall transport protocol is QUIC [42].

7 Conclusion

This thesis presented the requirements and design of the Streamr Network, a decentralized publish-subscribe system. The differences between its WebSocket and WebRTC implementations were described and evaluated experimentally. The details of the WebSocket, WebRTC and their underlying protocols were explored. To give context for the design and performance of the Streamr Network, previous research on P2P overlays systems was explored.

The Streamr Network's performance and qualities were measured and estimated with the use of theoretical analysis, simulations, emulations, and real-world experiments. The simulations showed that the tracker algorithm of the Streamr Network generates d-regular random graphs as intended. Based on this, theoretical analysis was used to predict the performance of the stream overlays. Emulation experiments performed on top of CORE were used to measure the bandwidth consumption of the network. The results showed that the system performs better than expected based on theoretical analysis under latency. Emulations were also used to compare the performances of the WebRTC and WebSocket stream overlays versions under packet loss. The results found that the performance of the WebRTC stream overlays was more predictable under packet loss rates up to 3%.

Real world experiments deployed over the internet in AWS showed that the performance of both the WebSocket and WebRTC versions scale logarithmically as a function of the number of nodes. Importantly, the WebSocket stream overlays were shown to have an unintentional bias towards the formation of low delay connections and thus were found to violate the tracker's instructions. The WebRTC stream overlays were shown to be d-regular random graphs as instructed by the tracker's algorithm. This is seen in the results of the experiments where WebSocket and WebRTC stream overlays had RDPs of 2.8 and 4.18 respectively at 2048 nodes. These numbers match with previous research on the performance of d-regular topology aware graphs and d-regular random graphs. Furthermore, the *mean message propagation delays* of streams can be estimated using ICMP ping delays between nodes at MAPE of 3.95% for WebSocket and 1.24% for WebRTC stream overlays. This shows that the measured increase in *mean message propagation delays* of the WebRTC stream overlays was not due to the transport protocol and instead WebRTC was validated as the better protocol for the performance requirements Streamr Network.

Based on the findings, a clear path on how to improve the performance of the stream overlays was found with TA. Once TA is introduced to the tracker algorithm, the WebRTC stream overlays are expected to outperform their WebSocket counterparts in terms of absolute *mean message propagation delays* and *relative delay penalty* based on the results of figure 5.8. All in all, this thesis shows that the WebRTC protocol is the better fit for the Streamr Network's requirements for *scalability (1)*, *low and predictable latency (3)* and *optimization for small payloads (4)*. This validates WebRTC as the transport protocol of choice for the Streamr Network's Brubeck release, which takes the first steps toward decentralization.

Bibliography

[1] M. Allman, V. Paxson, and E. Blanton. *TCP Congestion Control*. URL: https://tools.ietf.org/html/rfc5681.

[2] *Amazon Elastic Compute Cloud Documentation*. URL: https://docs.aws.amazon.com/ec2/index.html.

[3] *Auto Scaling groups*. URL: https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html.

[4] e. B. Y. Zhao Ling Huang. "Tapestry: a resilient global-scale overlay for service deployment". In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53. DOI: 10.1109/JSAC.2003.818784.

[5] 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. DOI: 10.1109/EC2ND.2008.7.

[6] 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*. DEBS '07. Toronto, Ontario, Canada: Association for Computing Machinery, 2007, pp. 2–13. ISBN: 9781595936653. DOI: 10.1145/1266894.1266898. URL: https://doi.org/10.1145/1266894.1266898.

[7] A. Barth. *The Web Origin Concept*. URL: https://tools.ietf.org/html/rfc6454.

[8] Ó. Blanco-Novoa, P. Fraga-Lamas, M. A Vilar-Montesinos, and T. M. Fernández-Caramés. "Creating the internet of augmented things: An open-source framework to make IoT devices and augmented and mixed reality systems talk to each other". In: *Sensors* 20.11 (2020), p. 3328.

[9] 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*. PODC '08. Toronto, Canada: Association for Computing Machinery, 2008, pp. 145–154. ISBN: 9781595939890. DOI: 10.1145/1400751.1400772. URL: https://doi.org/10.1145/1400751.1400772.

[10] *Boto3 documentation*. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

[11] E. Brewer. "CAP twelve years later: How the "rules" have changed". In: *Computer* 45.2 (2012), pp. 23–29. DOI: 10.1109/MC.2012.37.

[12] V. Buterin. *Ethereum Whitepaper*. URL: https://ethereum.org/en/whitepaper/.

[13] e. a. C. Holmberg. *Web Real-Time Communication Use Cases and Requirements*. URL: https://tools.ietf.org/html/rfc7478.

[14] M. Castro, P. Druschel, A. .-. Kermarrec, and A. I. T. Rowstron. "Scribe: a large-scale and decentralized application-level multicast infrastructure". In: *IEEE Journal on Selected Areas in Communications* 20.8 (2002), pp. 1489–1499. DOI: 10.1109/JSAC.2002.803069.

[15] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. "Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks". In: *Lecture Notes in Computer Science* 2584 (Oct. 2002).

[16] X. Chen and S. G. Murillo. *WebSocket Protocol as a Transport for Traversal Using Relays around NAT (TURN)*. URL: https://tools.ietf.org/id/draft-chenxin-behave-turn-websocket-01.html.

[17] 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*. DEBS '07. Toronto, Ontario, Canada: Association for Computing Machinery, 2007, pp. 14–25. ISBN: 9781595936653. DOI: 10.1145/1266894.1266899. URL: https://doi.org/10.1145/1266894.1266899.

[18] 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*. SIGMETRICS '00. Santa Clara, California, USA: Association for Computing Machinery, 2000, pp. 1–12. ISBN: 1581131941. DOI: 10.1145/339331.339337. URL: https://doi.org/10.1145/339331.339337.

[19] T. M. F. CIC. *Matrix Specification 1.0*. June 2019. URL: https://matrix.org/docs/spec.

[20] *Common Open Research Emulator (CORE)*. URL: https://www.nrl.navy.mil/Our-Work/Areas-of-Research/Information-Technology/NCS/CORE/.

[21] G. D'Angelo and S. Rampone. "A NAT traversal mechanism for cloud video surveillance applications using WebSocket". In: *Multimedia Tools and Applications* 77.19 (Oct. 2018), pp. 25861–25888. ISSN: 1573-7721. DOI: 10.1007/s11042-018-5821-z. URL: https://doi.org/10.1007/s11042-018-5821-z.

[22] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. URL: https://tools.ietf.org/html/rfc5246.

[23] W. Eddy. *TCP SYN Flooding Attacks and Common Mitigations*. URL: https://tools.ietf.org/html/rfc4987.

[24] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. "The Many Faces of Publish/Subscribe". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: https://doi.org/10.1145/857076.857078.

[25] I. Fette and A. Melnikov. *The WebSocket Protocol*. URL: https://tools.ietf.org/html/rfc6455.

[26] A. Ford and C. Raiciu. *TCP Extensions for Multipath Operation with Multiple Addresses*. URL: https://tools.ietf.org/html/rfc8684.

[27] N. Garg. *Apache kafka*. Packt Publishing Ltd, 2013.

[28] S. O. D. Gupta Abhishek. "Meghdoot: Content-Based Publish/Subscribe over P2P Networks". In: *Middleware 2004*. Ed. by H.-A. Jacobsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 254–273. ISBN: 978-3-540-30229-2.

[29] D. H. H. Le. "An efficient mechanism for mobility support using peer-to-peer overlay networks". In: *INDIN '05. 2005 3rd IEEE International Conference on Industrial Informatics, 2005.* 2005, pp. 325–330. DOI: 10.1109/INDIN.2005.1560397.

[30] R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. URL: https://tools.ietf.org/html/rfc2460.

[31] Y. Hirai. "Defining the Ethereum Virtual Machine for Interactive Theorem Provers". In: *Financial Cryptography and Data Security*. Ed. by M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson. Cham: Springer International Publishing, 2017, pp. 520–535. ISBN: 978-3-319-70278-0.

[32] 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.

[33] N. Jani and K. Kant. “SCTP performance in data center environments”. In: *Proceedings of SPECTS*. 2005.

[34] G. P. Jesi, A. Montresor, and M. van Steen. “Secure peer sampling”. In: *Computer Networks* 54.12 (2010). P2P Technologies for Emerging Wide-Area Collaborative Services and Applications, pp. 2086–2098. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2010.03.020. URL: https://www.sciencedirect.com/science/article/pii/S1389128610001672.

[35] V. John and X. Liu. “A survey of distributed message broker queues”. In: *arXiv preprint arXiv:1704.00411* (2017).

[36] J. Johnsen. *Peer-to-peer networking with BitTorrent*. http://web.cs.ucla.edu/classes/cs217/05BitTorre Dec. 2005.

[37] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web”. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, pp. 654–663.

[38] *Keeping Time With Amazon Time Sync Service*. URL: https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service/.

[39] A. Keranen and C. Holmberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. URL: https://tools.ietf.org/html/rfc8445.

[40] J. Kleinberg, M. Sandler, and A. Slivkins. “Network failure detection and graph connectivity”. In: *SODA*. Vol. 4. 2004, pp. 76–85.

[41] M. Kwon and S. Fahmy. “Topology-Aware Overlay Networks for Group Communication”. In: *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV ’02. Miami, Florida, USA: Association for Computing Machinery, 2002, pp. 127–136. ISBN: 1581135122. DOI: 10.1145/507670.507688. URL: https://doi.org/10.1145/507670.507688.

[42] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. "The quic transport protocol: Design and internet-scale deployment". In: *Proceedings of the conference of the ACM special interest group on data communication*. 2017, pp. 183–196.

[43] H. D. Le Hanh. "Parm: A Physically-Aware Reference Model For Peer-To-Peer Overlay Internetworking". In: *Journal of Interconnection Networks* 7 (Dec. 2006), pp. 451–474. DOI: 10.1142/S0219265906001806.

[44] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic Broadcast Trees". In: *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*. SRDS '07. USA: IEEE Computer Society, 2007, pp. 301–310. ISBN: 076952995X.

[45] C. Lesniewski-Laas. "A Sybil-Proof One-Hop DHT". In: SocialNets '08. Glasgow, Scotland: Association for Computing Machinery, 2008, pp. 19–24. ISBN: 9781605581248. DOI: 10.1145/1435497.1435501. URL: https://doi.org/10.1145/1435497.1435501.

[46] A. Loewenstern and A. Norberg. *DHT Protocol*. Jan. 2008. URL: http://bittorrent.org/beps/bep%5C_0005.html.

[47] e. a. M. Handley. *SDP: Session Description Protocol*. URL: https://tools.ietf.org/html/rfc4566.

[48] D. Madhuri and P. C. Reddy. "Performance comparison of TCP, UDP and SCTP in a wired network". In: *2016 International Conference on Communication and Electronics Systems (ICCES)*. 2016, pp. 1–6. DOI: 10.1109/CESYS.2016.7889934.

[49] S. V. Margariti and V. V. Dimakopoulos. "A study on the redundancy of flooding in unstructured p2p networks". In: *International Journal of Parallel, Emergent and Distributed Systems* 28.3 (2013), pp. 214–229.

[50] N. Masinde and K. Graffi. "Peer-to-Peer-Based Social Networks: A Comprehensive Survey". In: *SN Computer Science* 1.5 (Sept. 2020), p. 299. ISSN: 2661-8907. DOI: 10.1007/s42979-020-00315-8. URL: https://doi.org/10.1007/s42979-020-00315-8.

[51] M. Mathis and J. Mahdavi. *TCP Selective Acknowledgment Options*. URL: https://tools.ietf.org/html/rfc2018.

[52] M. D. Maymounkov Petar. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: *Peer-to-Peer Systems*. Ed. by P. Druschel, F. Kaashoek, and A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-45748-0.

[53] D. Mills, J. Martin, J. Burbank, and W. Kasch. “Network time protocol version 4: Protocol and algorithms specification”. In: (2010).

[54] J. Moy. *OSPF Version 2*. URL: https://tools.ietf.org/html/rfc2328.

[55] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck. “Blockchain”. In: *Business & Information Systems Engineering* 59.3 (2017), pp. 183–187.

[56] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. “Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation”. In: *IEEE/ACM Trans. Netw.* 8.2 (Apr. 2000), pp. 133–145. ISSN: 1063-6692. DOI: 10.1109/90.842137. URL: https://doi.org/10.1109/90.842137.

[57] J. A. Patel, É. Rivière, I. Gupta, and A.-M. Kermarrec. “Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems”. In: *Computer Networks* 53.13 (2009). Gossiping in Distributed Systems, pp. 2304–2320. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2009.03.018. URL: https://www.sciencedirect.com/science/article/pii/S1389128609001261.

[58] H. Pihkala and R. Karjalainen. *Unstoppable Data for Unstoppable Apps: DATAcoin by Streamr*. URL: https://s3.amazonaws.com/streamr-public/streamr-datacoin-whitepaper-2017-07-25-v1_1.pdf.

[59] E. R. Stewart. *Stream Control Transmission Protocol*. URL: https://tools.ietf.org/html/rfc4960#section-3.3.2.

[60] S. Ratnasamy and e. Francis. “A Scalable Content-Addressable Network”. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’01. San Diego, California, USA: Association for Computing Machinery, 2001, pp. 161–172. ISBN: 1581134118. DOI: 10.1145/383059.383072. URL: https://doi.org/10.1145/383059.383072.

[61] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. “Application-level multicast using content-addressable networks”. In: *International Workshop on Networked Group Communication*. Springer. 2001, pp. 14–29.

[62] T. Reddy and A. Johnston. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. URL: https://tools.ietf.org/html/rfc8656.

[63] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. URL: https://tools.ietf.org/html/rfc8446.

[64] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. URL: https://tools.ietf.org/html/rfc6347.

[65] M. Resende and P. Pardalos. *Handbook of Optimization in Telecommunications*. Jan. 2008, pp. 192–197. DOI: 10.1007/978-0-387-30165-5.

[66] J. Rosenberg and e. R. Mahy. *Session Traversal Utilities for NAT (STUN)*. URL: https://tools.ietf.org/html/rfc5389.

[67] D. P. Rowstron Antony. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Middleware 2001*. Ed. by R. Guerraoui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 329–350. ISBN: 978-3-540-45518-9.

[68] M. H. S. Ratnasamy. "Topologically-aware overlay construction and server selection". In: *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 3. 2002, 1190–1199 vol.3. DOI: 10.1109/INFCOM.2002.1019369.

[69] T. Saedi and H. El-Ocla. "Improving throughput in lossy Wired/Wireless networks". In: *Wireless Personal Communications* 114 (2020), pp. 2315–2326.

[70] P. Savolainen, S. Juslenius, E. Andrews, M. Pokrovskii, S. Tarkoma, and H. Pihkala. "The Streamr Network: Performance and Scalability". In: (). URL: https://streamr-public.s3.amazonaws.com/streamr-network-scalability-whitepaper-2020-08-20.pdf.

[71] 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*. Middleware '12. ontreal, Quebec, Canada: Springer-Verlag, 2012, pp. 271–291. ISBN: 9783642351693.

[72] B. Sikdar, S. Kalyanaraman, and K. S. Vastola. "Analytic models for the latency and steady-state throughput of TCP Tahoe, Reno, and SACK". In: *IEEE/ACM Transactions On Networking* 11.6 (2003), pp. 959–971.

[73] D. Skvorc, M. Horvat, and S. Srbljic. “Performance evaluation of Websocket protocol for implementation of full-duplex web streams”. In: May 2014, pp. 1003–1008. ISBN: 978-953-233-077-9. DOI: 10.1109/MIPRO.2014.6859715.

[74] P. Srisuresh. *IP Network Address Translator (NAT) Terminology and Considerations*. URL: https://tools.ietf.org/html/rfc2663.

[75] A. STEGER and N. C. WORMALD. “Generating Random Regular Graphs Quickly”. In: *Combinatorics, Probability and Computing* 8.4 (1999), pp. 377–396. DOI: 10.1017/S0963548399003867.

[76] M. R. Stoica Ion. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’01. San Diego, California, USA: Association for Computing Machinery, 2001, pp. 149–160. ISBN: 1581134118. DOI: 10.1145/383059.383071. URL: https://doi.org/10.1145/383059.383071.

[77] J. Stone, R. Stewart, and D. Otis. *Stream control transmission protocol (SCTP) checksum change*. Tech. rep. RFC 3309, September, 2002.

[78] M. Szydlo. “Merkle tree traversal in log space and time”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2004, pp. 541–554.

[79] A. Taivalsaari and T. Mikkonen. “A Roadmap to the Programmable World: Software Challenges in the IoT Era”. In: *IEEE Software* 34.1 (Jan. 2017), pp. 72–80. ISSN: 0740-7459. DOI: 10.1109/MS.2017.26.

[80] Theory.org. *Bittorrent Protocol Specification v1.0*. 2021. URL: https://wiki.theory.org/index.php/BitTorrentSpecification.

[81] Y. Tian, H. Shen, and K.-W. Ng. “Improving Reliability for Application-Layer Multicast Overlays”. In: *IEEE Trans. Parallel Distrib. Syst.* 21 (Aug. 2010), pp. 1103–1116. DOI: 10.1109/TPDS.2009.166.

[82] Tran Cong Hung and Tran Phu Khanh. “Analyze and Evaluate the performance of SCTP at transport layer”. In: *2010 The 12th International Conference on Advanced Communication Technology (ICACT)*. Vol. 1. 2010, pp. 294–299.

[83] *TRANSMISSION CONTROL PROTOCOL*. URL: https://tools.ietf.org/html/rfc793.

[84] M. Tuexen and e. R. Stewart. *Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets*. URL: https://tools.ietf.org/html/rfc8261.

[85] M. Tuexen and R. Seggelmann. *Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)*. URL: https://tools.ietf.org/html/rfc6083.

[86] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, and Y. Psaras. *GossipSub: Attack-Resilient Message Propagation in the Filecoin and ETH2.0 Networks*. July 2020.

[87] *WebRTC 1.0: Real-Time Communication Between Browsers*. URL: https://www.w3.org/TR/webrtc/.

[88] F. Weinrank, M. Becke, J. Flohr, E. Rathgeb, I. Rungeler, and M. Tuxen. "WebRTC Data Channels". In: *IEEE Communications Standards Magazine* 1.2 (2017), pp. 28–35. DOI: 10.1109/MCOMSTD.2017.1700007.

[89] *Wondernetwork*. URL: https://wondernetwork.com/pings.

[90] N. C. Wormald et al. "Models of random regular graphs". In: *London Mathematical Society Lecture Note Series* (1999), pp. 239–298.

[91] E. Zegura, K. Calvert, and S. Bhattacharjee. "How to model an internetwork". In: *Proceedings of IEEE INFOCOM '96. Conference on Computer Communications*. Vol. 2. 1996, 594–602 vol.2. DOI: 10.1109/INFCOM.1996.493353.

[92] H. Zhang, A. Goel, and R. Govindan. "Incrementally Improving Lookup Latency in Distributed Hash Table Systems". In: *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '03. San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 114–125. ISBN: 1581136641. DOI: 10.1145/781027.781042. URL: https://doi.org/10.1145/781027.781042.

[93] Y. Zhu and Y. Hu. "Ferry: A P2P-Based Architecture for Content-Based Publish/Subscribe Services". In: *IEEE Transactions on Parallel and Distributed Systems* 18.5 (2007), pp. 672–685.

[94] Y. Zhu, M.-Y. Wu, and W. Shu. "Comparison study and evaluation of overlay multicast networks". In: *2003 International Conference on Multimedia and Expo. ICME'03. Proceedings (Cat. No. 03TH8698)*. Vol. 3. IEEE. 2003, pp. III–493.

[95] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. “Bayeux: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination”. In: *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV ’01. Port Jefferson, New York, USA: Association for Computing Machinery, 2001, pp. 11–20. ISBN: 1581133707. DOI: 10.1145/378344.378347. URL: https://doi.org/10.1145/378344.378347.