



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Controlling Intelligent Prostheses Using 5G MEC

Scientific Students' Association Report

Author:

Gergely Gembela

Advisor:

dr. Gergely Mezei

2024

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
1.1 Related Work	2
1.2 Goals	2
1.3 Embedded Hardware Might Get There	3
1.4 Requirements	3
2 The Design and Specification of the Protocol	4
2.1 Inspiration and Alternatives	4
2.2 Sessions	5
2.3 Definitions	6
2.4 Message Types	7
2.4.1 Client-Initiated Control Messages	7
2.4.1.1 InitRequest	7
2.4.1.2 DisconnectRequest	8
2.4.1.3 StartTrainingRequest	8
2.4.1.4 StartInferenceRequest	8
2.4.2 Server-Initiated Control Messages	8
2.4.2.1 InitResponse	8
2.4.2.2 StateReport	9
2.4.3 Data Messages	9
2.4.3.1 Data	9
2.4.3.2 DataResponse	9
2.4.3.3 DataGuaranteed	9
2.4.4 Utility Control Messages	10
2.4.4.1 ErrorReport	10

2.4.4.2	Unknown	10
3	Implementation	11
3.0.1	Tech stack	11
3.0.1.1	Dependencies	11
3.1	Modules and Architecture	12
3.2	Points of Extensibility and Customization	13
3.2.1	New Message Types	13
3.2.2	The Server-side Executor Trait	13
3.2.2.1	The Python Server-side Executor	13
3.3	Lessons Learned	13
3.3.1	Windows, Linux and Scheduling	13
3.3.2	Tokio Tasks	14
3.3.3	Python	14
3.3.4	Asynchronous Processing	14
3.3.4.1	Timeouts	15
4	Benchmarks and Tests	16
4.1	Methodology	16
4.1.1	Devices and software	16
4.1.2	The UDPEcho Utility	17
4.1.3	Further Validation Using IPerf and ICMP Messages	17
4.1.4	The Measure Module	18
4.1.5	Graphs and Tables	18
4.2	Results	19
4.2.0.1	Baseline	19
4.2.1	5G Benchmarks	19
4.2.1.1	5G SA	19
4.2.1.2	5G NSA (Vodafone)	20
4.2.2	Wi-Fi	23
4.2.2.1	Wi-Fi 5	23
4.2.2.2	Wi-Fi 6	25
4.2.3	Measurements Besides the Shown Results	26
4.2.4	Comparison of Technologies	26
5	Conclusions and Future Work	28
5.1	Conclusions	28
5.2	Future Work	29

5.2.1	Guaranteed Transmission	29
5.2.2	Analysis of Historical Data	29
5.2.3	Parallel Execution of Models	29
5.2.4	IPv6 and Path MTU Discovery	30
5.2.5	Custom Payloads	30
5.2.6	5G and Beyond 5G Technologies	30
5.2.7	Integration With a Physical Prosthesis	30
5.2.8	Embedded Targets	30
5.2.9	Deployment and Scaling	31

Bibliography	33
---------------------	-----------

Appendix	36
-----------------	-----------

A.1	5G SA	36
A.1.1	Different Payload Sizes	42
A.2	5G NSA	46
A.2.1	Different Payload Sizes	52
A.3	Wi-Fi 5	56
A.3.1	Different Payload Sizes	62
A.4	Wi-Fi 6	66
A.4.1	Different Payload Sizes	70
A.5	Comparison of Technologies	74

Kivonat

Napjainkban az intelligens protézisek vezérléséhez szükséges algoritmusok futtatása általában lokálisan, a protézisbe integrált vezérlő segítségével történik. A vezérlőalgoritmusok komplexitását korlátozza a rendelkezésre álló számítási teljesítmény, ezért nem alkalmazhatóak tetszőlegesen nagy számítási igényű algoritmusok a felhasználó szándékának megállapításához, a korlátozott pontosság viszont rontja a felhasználói élményt.

Modern hálózati technológiák és edge cloud megoldások használatával a rendelkezésre álló számítási kapacitás nagyságrendekkel nagyobb lehetne, mint beágyazott rendszerek esetében. Pontosán becsülhető késleltetés és overhead esetén lehetővé válna összetettebb, nagyobb számítási igényű, pontosabb vezérlést nyújtó algoritmusok futtatása a szenzorok által gyűjtött adatok feldolgozásakor, a hálózat által hozzáadott késleltetés árán.

A kutatóprojekt célja annak megállapítása, hogy az 5G hálózat, és hálózati technológiai képesek-e a vezérléskor elvégzendő számítások edge-cloud alkalmazásokba történő áthelyezéséhez megfelelő feltételeket és háttérrel biztosítani.

A dolgozat bemutatja a projekt részeként elkészített keretrendszert amely képes közel valós idejű számítások áthelyezésére IP hálózat segítségével, valamint a hálózatok, és a protokoll tesztelésére létrehozott eszközöket. A rendszer elsődleges alkalmazási területe az intelligens protézisvezérlés, ahol EMG adatok gyűjtése és feldolgozása során megállapításra kerül egy a felhasználó által kiváltani kívánt mozdulatosztály, és a protézis vezérléséhez szükséges vezérlő értékek. A protokoll tervezésekor szempont volt, hogy a rendszer egyéb, hasonló jellemzőkkel rendelkező vezérlési feladatokra is adaptálható legyen. Méréseken és kísérleteken keresztül bemutatásra kerülnek a keretrendszer képességei, és a jelenleg elérhető hálózatok által nyújtott lehetőségek.

Abstract

Currently, algorithms required to operate intelligent prostheses are usually executed locally, using the embedded system integrated into prostheses. However, the limited computational capabilities of such devices usually constrain the complexity and thus the accuracy of the algorithms that determine the intention of the user, resulting in a sub-optimal user experience.

Modern networking technologies in combination with edge cloud solutions offer significantly more raw computational power (in comparison with embedded solutions). With predictable latency and low overhead, more robust and accurate algorithms could be applied for processing data collected by sensors, thus providing more accurate control, at the cost of the latency introduced by the network.

The goal of my research project is to determine whether 5G networks and edge cloud solutions are viable as the background for offloading parts of a control system to edge cloud applications and address the challenges of offloading control in time-critical settings with a relatively high-frequency input such as electromyography (EMG) sensors.

This paper describes a framework for offloading near real-time computations using an IP network along with a set of tools for benchmarking networks and the protocol itself in different network conditions. The control of intelligent prostheses is an application the framework is specifically tailored to. EMG data is collected and used to determine the intended action, with the accuracy depending on the complexity of a classifier algorithm. Other applications related to control systems with similar requirements were also kept in mind during the design phase. The capabilities of different networking technologies in combination with the framework are shown through the results of various benchmarks and experiments.

Chapter 1

Introduction

Intelligent prostheses are usually controlled with algorithms executed on an embedded controller integrated into the prostheses. With recent developments in wireless networking technologies and (the) notable developments connected to edge cloud computing, offloading the processing of EMG signals to the edge cloud could help minimize local energy consumption and provide a better user experience with better accuracy than previously possible. Multi-access Edge Computing (MEC) is a European Telecommunications Standards Institute (ETSI) Industry Specification Group (ISG) standard [1] that standardizes edge cloud computing, often deployed and used together with 5G and Beyond 5G (B5G) networks. The standard extends to other access networks such as WLAN or Ethernet. MEC offers low latency and high bandwidth compared to traditional, centralized cloud services provided by Microsoft, Amazon, Oracle, IBM and more.

Several surveys are available focusing on the performance, dependability and possibilities of MEC [19] [12]. MEC is stated as the enabler of ultra-reliable and low-latency communications (URLLC), which could lift the solution described in this paper up a level, but MEC, when used together with regular and readily available networks and devices without the guarantees of URLLC opens up plenty of new options.

3GPP is working on a family of devices that implement only parts of the 5G standards referred to as RedCap¹ (Reduced Capability) devices. The technology is also referred to as NR-Light. The offloading of EMG processing fits the specifications listed for safety-related industrial wireless sensors and might provide a lower-cost alternative to URLLC solutions.

5G can be deployed either as a standalone (SA) network in which all components (e.g. the core, the radio, and the infrastructure) are 5G devices and as a non-standalone (NSA) network, in which 5G features are built on top of 4G infrastructure. Unfortunately, most deployments are non-standalone as NSA 5G has no problem meeting the expectations of an average user (i.e. phone/mobile data users) and NSA is a natural step in the migration process from 4G to 5G. However, NSA networks lack key 5G functionalities, and the lack of 5G SA networks slows the deployment of technologies like RedCap².

Intelligent prostheses usually measure EMG data, and based on the measured values a classifier algorithm determines the intended type and intensity of a given intent [22]. Although embedded SoCs and AI accelerators are becoming more powerful and energy-efficient by the day, the gap between low-power solutions and edge cloud servers remains significant.

¹<https://www.3gpp.org/technologies/nr-redcap-glimpse>

²<https://www.fierce-network.com/broadband/lack-5g-standalone-networks-slows-redcap-deployment>

Embedded solutions usually use feature extraction before the classification of EMG data (e.g. [16]) which (in addition to complexity constraints) often results in less accurate classification, but the processing gets more efficient in terms of computational requirements.

In a MEC application, the application of a more complex classifier model (or even multiple models) is possible with an execution time comparable to that of embedded models. MEC applications also have enough resources to store measured data for further analysis. To achieve the best possible user experience, the ultimate goal is to minimize latency and maximize battery life and *accuracy*.

1.1 Related Work

Research in the field of edge computing and offloading computations to the cloud or edge cloud environments (at least for now) is more focused on making the decisions whether to offload a task or not, and the analysis of those decisions (mainly focusing on cost both in terms of latency, and resources). Offloading control-related computations to the edge cloud fits the "Edge Native Design" paradigm outlined in [6]

AR/VR and computer vision-related applications are a prime use case of edge computing as both its latency and bandwidth requirements push networks to the limits, and rule out the use of traditional cloud computing, and research seems to be focused mostly on bandwidth/throughput and QoS concerns [13].

Industrial IoT applications are often combined with private 5G networks [7] and solutions are focused on a generalised architecture rather than minimal overhead [26].

The integration of edge computing and machine learning is referred to as Edge intelligence [17], I found this to be the field of research closest to mine.

1.2 Goals

In a sense, my task was simpler compared to those presented in Related Work, because the domain is specific to EMG processing, and the bandwidth requirements are much less significant. Data to be sent periodically does not vary in size, which makes the bandwidth requirement of the communication well-predictable. The goal of the research project is to determine whether current networking technologies in combination with edge cloud solutions are a suitable alternative to on-device computations and to prove that 5G (and other current-gen wireless) networks are capable of offloading the compute-intensive parts of a control loop. The objective we set to be achieved until the release of this paper was the creation of an environment which makes the analysis of the suitability of these technologies possible through measurements in real-world conditions.

To get closer to the ultimate goal, a protocol, the subject of this paper was developed for low-latency EMG data transmission, and a goal to achieve a maximum RTT of 150 ms (including time dedicated for processing) was set.

1.3 Embedded Hardware Might Get There

Embedded SOCs are getting more powerful and energy efficient by the day while chip manufacturers seem to focus on *performance per watt*³ more than ever. Thanks to industries such as AR/VR, embedded IoT and the TinyML⁴ paradigm companies invest more and more (funds and) resources into developing mobile/embedded technologies with dedicated hardware accelerators that make embedded AI solutions more efficient. In the future, for some workloads the execution of more robust and energy-efficient models locally [27] might become a viable alternative to network-based offloading.

While the protocol and the server-side classifier might serve only as a transient solution for controlling intelligent prostheses, the findings, measurements, the protocol and the offloading paradigm might remain relevant in use cases that rely on large or frequently changing server-side databases or models. The experience I gained with using 5G in real-time and/or control-oriented applications applies to a wide range of other challenges.

1.4 Requirements

In the following subsections, I outline the requirements that served as a base while developing the application layer network protocol that facilitates offloading of control to a MEC environment. Although the requirements originally derive from the main use case (EMG processing), it might also be applicable to use cases with similar networking requirements.

- **Minimal overhead, up-to-date state:** The nature of the task requires the protocol to deliver data in both directions as quickly as possible, with as little overhead as possible. The requirements (and expectations) are different for communications that control the protocol itself and that transmit data, therefore two kinds of messages are to be distinguished and might be treated differently.
- **Extensibility:** The protocol should be designed to be easily extensible with new message types, tests, benchmarks and server-side modules that handle the received messages. To leverage compiler optimizations and keep the code base as simple as possible, native modules are not dynamically extensible.
- **Security:** Communications over the protocol must be protected against eavesdropping, tampering, message forgery and replayability.
- **Reliability:** In use cases such as controlling intelligent prostheses, reliability is a key concern. The protocol must be able to detect connection errors and needs to make the use of a locally executed fallback classifier possible. Note that we do not (and cannot) expect the same level of efficiency when processing the data locally, but this is hardly an issue, as the chances of requiring the use of the fallback classifier should be minimal under normal circumstances.
- **Configurability:** The protocol aims to support different kinds of EMG feature vectors with a configurable number of electrodes, ADC resolution (in most cases this comes down to float precision), and window size (the number of vectors sent as a single message).

³<https://newsroom.arm.com/blog/performance-per-watt>

⁴<https://newsroom.arm.com/blog/tinymml>

Chapter 2

The Design and Specification of the Protocol

NWEmg is a protocol designed to make offloading EMG processing from intelligent prostheses to the edge cloud simple, efficient and reliable. The protocol was designed to achieve the best possible performance with the lowest possible overhead both in terms of computations and network resources.

While the guarantees provided by the URLLC 5G communications would make the protocol seem extremely robust, the protocol does not require the use of URLLC, and was designed to operate in all common modern wireless networking environments (5G/Wi-Fi6), and a fallback mechanism is expected to be available locally. Most classifier algorithms are stateless, therefore packet (and as a direct result message) loss is tolerable to a certain extent. The potential reordering of messages should be accounted for by the implementation.

The protocol exchanges binary data with messages of predefined formats specified in Message types (rather than using a semi-structured data interchange format like JSON or BSON). The byte ordering of data types used by the protocol is little-endian (LE) regardless of the platform.

An implementation was created for the protocol that adheres to the specification of the protocol and is presented in Implementation.

2.1 Inspiration and Alternatives

Before designing a new protocol, a natural step was to look into existing protocols that could be applied to EMG data transmission.

MQTT¹ and CoAP [25] were examined as alternative application layer protocols. MQTT is used in multiple papers in the field of MEC offloading, but it uses transmission control protocol (TCP) which is impractical due to the requirement *Minimal overhead, up-to-date state*, as in real-time and near real-time use cases it might suffer extra latency due to head-of-line blocking[24]. In the case of CoAP, the extra functionality it would provide is not worth the overhead compared to native UDP.

Stream Control Transmission Protocol (SCTP) [15], a message-oriented, but connection-based transport layer protocol provides an ideal subset of features of both UDP (no HOLB,

¹<https://mqtt.org/mqtt-specification/>

optional reliability) and TCP (connection-oriented). My reason for not choosing SCTP was that most home routers which need to perform NAT in case of IPv4 traffic still do not support SCTP (this unfortunately includes the customer premise equipments (CPEs) used with the private 5G network used in Benchmarks and Tests), and while SCTP over UDP would have been an option, with some additional work UDP alone can meet the requirements of the protocol.

The takeaway was that even though solutions exist both for IoT communications (CoAP and MQTT) and real-time streaming (RTP) none of these target low-bandwidth and low-latency use cases such as the transmission of EMG data, and a protocol specifically designed for this purpose could achieve lower overhead both in terms of additional data transmitted, and computational requirements.

Using UDP has the advantage that it has the lowest possible overhead and even the application layer protocol can be kept rather simple as the exact structure/size of datagrams is always known before transmission/reception.

The drawbacks of using UDP are that the protocol needs to handle connections, ensure the detection of reordering, packet loss or multiple reception and guarantee the consistency of the messages (as in IPv4, the UDP checksum is optional, and the 16-bit checksum is rather weak).

As per UDP Usage Guidelines [11] Datagram Transport Layer Security (DTLS) is used to secure communications over the protocol.

2.2 Sessions

Even though UDP is connectionless, the protocol uses the concept of sessions. Sessions (in this case) are a temporary, stateful series of communications between the client and the server. Sessions are identified by 128-bit UUIDs but sessions are directly associated with the IP addresses and ports of clients.

The possible states and legal state transfers are shown with black arrows in the state machine diagram. State transfer requests are idempotent and illegal state transfers must always result in an error reply.

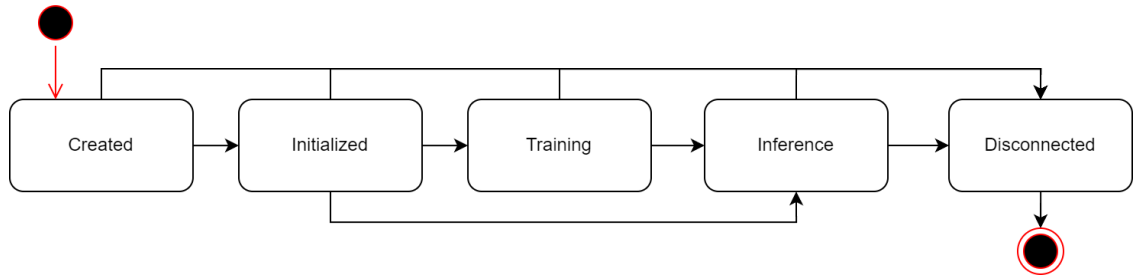


Figure 2.1: The state machine representation of a session's state

The following paragraphs describe the lifecycle of a session over the *NWEmg* protocol.

Both the server and the client are initialised with the state *Created*. This state indicates that the processes were started, but no communication occurred, and the server is waiting for a client to connect.

The client sends an *InitRequest* message to the server to establish a new session. When the init message is received by the server, it transfers its state to a hidden *Initialising*

state. This state is necessary because initialization of the classifier model, or other setup actions that happen during the *Created->Initialized* state transfer could last longer than the timeout value, and initialization (as all other state transfers) can only happen once regardless of the number of requests.

After initialization, the client may proceed to the states *Training* or *Inference*. If the client establishes a new session with the server, the user has to train the classifier model, and the session must proceed to *Training* state. If a context with a trained model can be restored, for the user (e.g. in case of reconnection caused by a network error) the *Initialized->Inference* state transfer is legal and should be used.

In both *Training* and *Inference* states the protocol transmits and accepts messages related to EMG processing. Data-related communications happen asynchronously, and data messages may not alter the state of either party.

The client may terminate the session in any state. When the server receives the request, the server informs the client that the session state is *Disconnected*, the session is terminated, and all server-side resources related to the session are freed up.

The state transfer *Initialized* to *Inference* is legal because upon reconnection the server might be able to restore a previous session or training set (or even a trained model depending on the implementation).

2.3 Definitions

Message: A message is the unit transmitted by the protocol. On IPv4 usually 1 IP packet, on IPv6 strictly one IP packet.

Numeric types: The specification uses the Rust type names² as they concisely represent numeric types. For the following chapters, the format is (u|i|f)<number of bits>, where u means unsigned integer, i means signed integer, and f represents a floating point value type. (E.g. u64 is a 64-bit unsigned integer, and f32 is a 32-bit floating point number.)

Version triplet: At first glance a SemVer version, but in reality a triplet of u64 values: major, minor, and patch version numbers. When used as a string, the major, minor and patch values are separated by dots (e.g. "0.4.2").

Payload or message types: Each message type is introduced starting with a table, showing the fields of a structure along with their types. As an example, the table describing *version triplet* would be:

```
+-----+-----+-----+
| major: u64 | minor: u64 | patch: u64 |
+-----+-----+-----+
```

This format was chosen as it represents structures more efficiently than pseudocode structure definitions, but unlike tables in RFCs rows are not of a fixed size in bytes.

The protocol encapsulates all messages into an NWEmgDatagram structure. The structure represents the data in its final format that is sent over the DTLS and UDP layers.

```
+-----+-----+-----+-----+
| type: DatagramType | id: u32 | message: u8[1..MMS] | checksum: u32 |
+-----+-----+-----+-----+
```

²<https://doc.rust-lang.org/beta/book/ch03-02-data-types.html>

The maximum message size (MMS) is determined by the configured maximal UDP datagram size. The default maximal datagram size the protocol supports is 1500 bytes, as path maximum transmission units (MTUs) are typically not more than 1500 bytes.

DatagramType is a one-byte enumerated type, providing unified message type values. Message types are specified in the next section. New message types may be added in later versions, but the 1-byte type identifier is far from exhaustion.

The protocol uses 32-bit IDs that wrap on overflow by design. This ID length is used, as even at a relatively high (1000 messages/s) message rate, it would still take the counters more than 1000 hours to wrap. If sessions need to be reconstructible (for analysis or debugging purposes), limiting the session length to around 1000 hours does not constrain the protocol (at least in its current use case).

The sequence numbers provided by the DTLS layer are not used (in upper layers), as *NWEmg* sessions are not bound to DTLS sessions.

2.4 Message Types

The protocol has two different kinds of messages, control messages, and data messages. Control messages can be further divided into three categories based on the possible sender: (i) client (ii) server (iii) both parties.

All messages (regardless of their type) are sent through the same UDP sockets, but depending on their type the way a given message is handled might be entirely different. The implementation should be capable of processing multiple messages asynchronously, regardless of the type of incoming messages.

Control messages carry information required to operate the protocol itself, ensure that the client and server-side state is synchronized, and take care of error handling. Control messages are prone to packet loss, thus control messages should be resent by the client in case the reply to the request times out. Only the client should send messages repeatedly, control messages have idempotent effects on the server's state, therefore with each client-initiated retransmission results in a new reply being sent to the client.

To help implementations reduce or even eliminate the need for dynamic allocations, all messages are either of a fixed length or have their maximum sizes specified. The only message type that has a non-fixed size is the *Data* message, but a safe upper limit to its length always exists.

2.4.1 Client-Initiated Control Messages

These messages must be sent from a client. When a client receives a client-initiated message, the result should be an *Invalid Message* error.

2.4.1.1 InitRequest

```
+-----+-----+-----+
| version: Version | no_items: u32 | no_bytes_item: u32 |
+-----+-----+-----+
| batch_size: u32 | user_id: uuid | prev_session_id: uuid |
+-----+-----+-----+
```

The client sends an `Init` message when it wishes to establish a session. The message contains the version triplet of the client. The server sends a *mismatched version* error if the client's version is not compatible with the server. The following 3 values specify the size and format of data messages for the entire length of the session. (Dynamic reconfiguration might have benefits in the longer run, more on this in 5G and beyond 5G technologies).

The client ID field identifies the user, and the `previous_session_id` optionally holds the id of a session to be restored if a new session is initiated due to a network error or a previous failure.

Control messages should be sent using a (large enough) timeout that allows the server to complete long-running initialisation or clean-up tasks such as loading or training a complex classification model.

2.4.1.2 DisconnectRequest

```
+-----+
| reason: u32 |
+-----+
```

The client must send a disconnect request when it is shutting down or is in a state in which it won't send any further messages in the session. The disconnect request has a single field, a 32-bit ID reserved for disconnect reasons. Currently, the reason field is only used for logging but it could be used to inform the server to retain or instantly remove the resources allocated for a specific session.

2.4.1.3 StartTrainingRequest

The client must send a *StartInferenceRequest* when it needs the server to enter training mode. Currently, it has an empty payload, see Custom Payload for ideas on increasing its flexibility. This message type can only be sent in state *Initialized*.

2.4.1.4 StartInferenceRequest

The client must send a *StartInferenceRequest* when it needs the server to enter inference mode. Currently, it has an empty payload, see Custom Payload for ideas on increasing its flexibility. This message type can only be sent in states *Initialized*, or *Training*.

2.4.2 Server-Initiated Control Messages

These messages must be sent from a server. When a server receives a server-initiated message, the result should be an *Invalid Message* error.

2.4.2.1 InitResponse

```
+-----+-----+
| version: Version | session_id: Uuid |
+-----+-----+
```

The response sent to an *InitRequest*. Informs the client that the server-side state transfer to *Initialized* was successful, and the server is ready to accept further messages. The version field contains the version of the server, and a 128-bit session ID is generated for each session.

2.4.2.2 StateReport

```
+-----+-----+
| from: NWEmgState | to: NWEmgState |
+-----+-----+
```

The message used for state synchronization. The client follows the server-side state upon receiving a StateReport message. The message contains the state in which the state transfer was initiated (from) and the state after the state change (to).

State change messages with identical *from* and *to* states should be used to indicate when a state change request did not affect the server-side state. This might be required in various situations i.e. the loss of the initial response.

State transfer messages are not necessarily the result of StateChangeRequests.

2.4.3 Data Messages

These messages handle the transmission of EMG data and classification results.

2.4.3.1 Data

The message contains the elements of the data vector packed without padding with LE byte ordering. Requirements for delivering data messages are unique, with the most important aspect being the up-to-date delivery of control information. Data messages are never retransmitted (*DataGuaranteed* messages should be used when the loss of messages can not be tolerated).

2.4.3.2 DataResponse

```
+-----+-----+-----+
| originator: u32 | velocity: f32 | action id: u8 |
+-----+-----+-----+
```

The response sent to Data and DataGuaranteed messages. Contains information required to control the prostheses. The originator field is used to connect the response to the request (and as a result detect reordering). This message type is specific to EMG processing, the *action_id* field specifies the result of the classification, along with a corresponding velocity.

2.4.3.3 DataGuaranteed

The message kind carrying data semi-reliably. This message kind is an optional element of implementations. The delivery of these messages is guaranteed using redundancy based on confirmations.

When an implementation does not provide this functionality, error reports should inform the client that the functionality is not available.

2.4.4 Utility Control Messages

Both the client and server may send and/or receive utility messages.

2.4.4.1 ErrorReport

```
+-----+-----+-----+
| code: u8 | length: u16 | message: [u8; 1024] |
+-----+-----+-----+
```

The message carries information regarding an error that may or may not be fatal to the session. Error reports should be of the following kind:

- Invalid message
- Critical error
- Invalid state transfer
- Mismatched version
- Custom error

Error reports allow a variable length (max. 1024 byte long) message to be sent detailing the error. The length field holds the length of the message, the message field holds the UTF-8 encoded bytes of the error message.

2.4.4.2 Unknown

Under normal circumstances, it is impossible for the protocol to send or receive a message of type *unknown*, however, it seemed logical to create a message type that represents all messages received with an unknown message type.

Reasons for a server or a client receiving unknown messages could be undetected data corruption (though the chances of that happening are extremely low provided the 32-bit checksum), human error, or mismatched protocol versions (in case a new message type is introduced on one side, but not the other), but this case is accounted for during initialisation, as a compatibility check occurs.

The reception of messages of *unknown* kind should be treated as errors.

Chapter 3

Implementation

To validate, benchmark and test the protocol, I created an implementation that can run on the most popular platforms (Linux, Windows, MacOS). The implementation can interface with an existing framework for benchmarking the control of intelligent prostheses, written in Python. The framework is the product of ongoing research conducted by another research group and is not publicly available yet.

3.0.1 Tech stack

The language I chose for the implementation is Rust. Rust guarantees memory safety without the use of a garbage collector, or the excessive use of smart pointers. Even though Rust is advertised to have zero-cost abstractions and (in terms of overhead) relatively cheap memory safety, the ownership model has quite a significant impact on the final implementation (/the way problems are solved/).

Unfortunately, I found software development in Rust, quite slow and sometimes even frustrating. The borrow checker has limitations that (even with logically sound exclusive mutable access) can not trivially be overcome. Regardless of these limitations, the resulting code is memory-safe, and the ownership system makes concurrent programming manageable¹. Among others, these two advantages alone would be enough to make me choose Rust again for a similar project in the future.

A compiler for Rust-based ² critical systems is also available.

3.0.1.1 Dependencies

The implementation is built on top of the Tokio³ runtime, an asynchronous runtime created for writing efficient and reliable network applications.

The implementation uses the OpenSSL⁴ DTLS 1.2 [20] implementation (OpenSSL does not support DTLS 1.3 as for now) with the Rust wrapper Tokio-Openssl⁵, as DIY security is not an option in the main use case.

¹<https://doc.rust-lang.org/book/ch16-00-concurrency.html>

²<https://ferrocene.dev/en/>

³<https://tokio.rs/>

⁴<https://github.com/openssl/openssl>

⁵<https://github.com/tokio-rs/tokio-openssl>

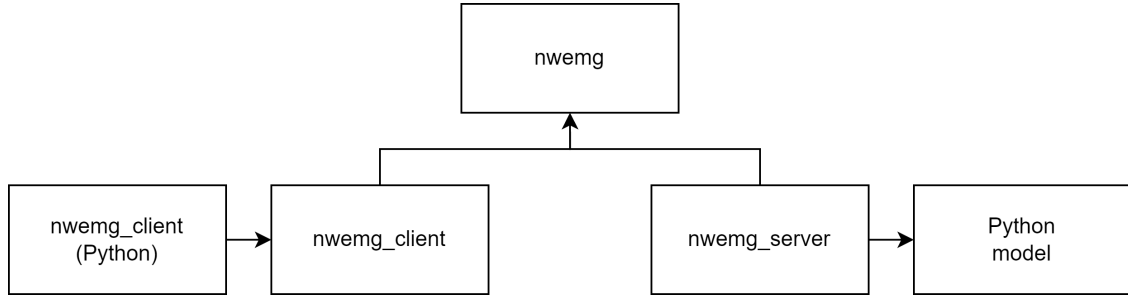


Figure 3.1: The module hierarchy of the implementation

PyO3⁶ is a library/toolset for creating native extension modules for Python and calling Python code from a native Rust process. The implementation uses both alternatives, as the client is built both as a Python Wheel and a Cargo crate, and the server is capable of executing a Python module as a classifier.

3.1 Modules and Architecture

While the architecture includes modules written in Python, the native components are fully featured and may be used without the need for executing non-native code, making the implementation suitable for use with embedded systems, or even microcontrollers.

There are three main native modules:

- `nwemg`: The library that contains the shared components.
- `nwemg_client`: The client library, built both as a Rust crate and as a Python library. This library could be consumed from all languages that support the C FFI, as it can be built and loaded as a C dynamic library. The client library contains the client-side benchmarking tools, of which a more detailed description is available in The Measure Module
- `nwemg_server`: An executable, that runs the server-side logic of the protocol. A heavily multi-threaded, high-performance and configurable server for processing EMG data.

The use of Python is optional, the two Python components are:

- Python client: a native Python module built with PyO3 that exposes native functions required to operate the protocol to be called from Python.
- Python server-side executor: an executor that loads a Python module on the server. (More details in The Python Server-side Executor.)

⁶<https://github.com/PyO3/pyo3>

3.2 Points of Extensibility and Customization

3.2.1 New Message Types

The implementation was created with extensibility in mind. New message types can be added by creating a `MessageType` entry, a message-specific payload and its serialization logic in the *nwemg* core library.

3.2.2 The Server-side Executor Trait

Rust traits are similar to C#/Java interfaces or C++ concepts. To customize the behaviour of the server, one needs to create an implementation of the server-side executor trait. The trait provides the following interception points:

- Global initialization
- Per-session initialization
- State transfers
- The reception of data

Implementations were created for network benchmarking (no processing, instantaneous reply), native classification (less flexibility, maximal performance), and Python code execution (maximal flexibility).

The methods of executors are called from tasks spawned for each message, therefore interception points might be called (potentially concurrently) from different threads.

3.2.2.1 The Python Server-side Executor

An executor implementation was created using PyO3 that loads a Python module. The interception points listed above are all available as Python functions. All Python dependencies (e.g. TensorFlow, PyTorch, etc.) are available. The Python executor makes testing various models easier, and when deployed, a native version of the best models can be created as separate server-side executors.

3.3 Lessons Learned

In this chapter, I summarize the most important experiences that directly influenced either the implementation or the benchmarking process.

3.3.1 Windows, Linux and Scheduling

While the implementation is cross-platform, there were several differences and obstacles while testing using different platforms. One of these differences was the standard resolution of the Windows timer tick. The default resolution is 15.6 ms⁷ which results in inconsistent

⁷<https://randomascii.wordpress.com/2020/10/04/windows-timer-resolution-the-great-rule-change/>

measurements and scheduling as it directly affects the behaviour of both Rust and Python schedulers. As an example, calling `sleep` in Tokio with a value lower than the timer resolution results in the future sleeping for an inconsistent duration, which might be almost 15.6 ms if the sleep is requested right after a timer interrupt. In contrast, on Linux, the default is usually 1 ms⁸. The default values can be changed, but the minimum value on Windows is still not low enough to provide the accuracy needed for measurements. For this reason, during benchmarks a task with busy-waiting is used regardless of the platform to ensure that messages are sent precisely on schedule.

3.3.2 Tokio Tasks

In Tokio, tasks are the unit of execution. Tasks are submitted for execution to a scheduler, which executes them concurrently while it has work to do potentially (but not necessarily) using multiple threads.

I had limited experience with Tokio before implementing the protocol exclusively using Tokio tasks. While Tokio handles asynchronous execution intuitively, unlike futures in other languages, Rust/Tokio futures do not progress unless polled (or when the poll method of a task turns out to be blocking⁹).

Spawning¹⁰ a task in Tokio is a lightweight operation as it requires a single allocation of 64 bytes of memory, and spawning even millions of tasks is not an anti-pattern.

Both the client and the server spawn tasks to process messages, allowing most of the processing (on the server this includes sending a reply) to happen concurrently.

3.3.3 Python

In a time-critical use case, the use of Python seems somewhat illogical due to its limited performance, but with true asynchronous processing the performance impact of running Python might be outweighed by the simplicity and the ease/speed of iterations. The implementation exposes native functions to Python and provides an API that makes the implementation of classifier models possible in Python.

Unfortunately, there was more to this issue. The main issue with the Python server-side executor is the Python Global Interpreter Lock (GIL), which prevents multiple threads from taking control of the interpreter at once¹¹. Single-threaded execution hardly fits into an environment built to leverage asynchronous execution. With Python 3.13 (released on October 7, 2024) the use of GIL was made optional making true multithreading possible, but components and libraries that assume the use of the GIL will take time to adapt.

3.3.4 Asynchronous Processing

The implementation processes all messages asynchronously, both on the server and on the client. As described in Inspiration and Alternatives UDP provides no guarantees to datagram delivery, therefore the implementation handles the detection of lost, out-of-order and duplicated datagrams with the use of multiple atomic counters.

⁸<https://timerresolution.net/what-is-the-default-timer-resolution-in-linux/>

⁹<https://rustmagazine.org/issue-4/how-tokio-schedule-tasks/>

¹⁰<https://tokio.rs/tokio/tutorial/spawning>

¹¹<https://wiki.python.org/moin/GlobalInterpreterLock>

After messages are decrypted, new tasks are spawned for each data message and even control messages that are not trivial to process.

The asynchronous processing of messages poses unexpected challenges, as on scheduled networks (such as 5G NR or Wi-Fi 6) there is a chance that messages are buffered/queued, and multiple datagrams are received either out of order or with such little time difference that the asynchronous processing of each message on the server might result in the tasks completing out of order.

To circumvent this problem, the implementation uses atomic counters, and tasks processing the messages are halted until a timeout (currently 3 ms) is reached, or the message with the expected sequence number is received.

The current implementation of the client spawns a Tokio task for each message sent that handles the reception and the processing of the reply. The first version of the client assumed that each task would be handled in the order they were submitted, but this is not the case, and even if it was, this method would not have handled packet loss or reordering.

To solve this problem, a custom task type was created that waits for its specific reply to arrive, or times out after reaching the configured data message timeout (500 ms by default).

Unfortunately, OpenSSL does not provide a fully asynchronous DTLS implementation/API, and synchronization is required for the encryption and decryption stages.

3.3.4.1 Timeouts

The protocol requires different, configurable timeout values. The following timeout types are specific to the client and represent round-trip times (RTTs).

- Control messages in general may require and/or tolerate a much higher timeout value, as state transfers and other control actions are not required to be optimized to provide instantaneous responses. The default timeout value for control messages is 500 ms, but an increase might be required if more complex models require more time to configure/fit/load.
- Data messages are treated differently. Generally, data messages are configured to time out after 500 ms just like control messages, but when a newer message was already accepted, the message is discarded.
- Optionally, other timeout values might need to be introduced for different state transfers.

Server-side DTLS sessions need to time out fast enough to allow clients to reconnect in case of a network failure or change but still leave time for transient errors to resolve. Currently, DTLS sessions are destroyed after 10 seconds.

Chapter 4

Benchmarks and Tests

The following chapters present the methodology for benchmarking the protocol/different access networks and also the results of various benchmarks.

4.1 Methodology

4.1.1 Devices and software

To minimize the skew of results caused by resource constraints of different configurations, devices used in each case were significantly more powerful than what would be required during the regular operation of the protocol.

The hardware configuration of the main client:

- Intel Core i5-10400
- 32 GB 3600 MHz DDR4 memory
- Realtek RTL 8111 GbE Network controller for measurements in which ethernet was used to connect to a CPE
- Intel Dual Band Wireless-AC 3168NGW WiFi 5 adapter used for Wi-Fi 5 measurements

For monitoring purposes, a different NIC was used. The routing table was set up for the client to use the specified interfaces only for traffic relevant to the measurements.

In the case of measurements that featured a phone, I used a Poco X6 5G¹.

I used UserLAnd² to run Ubuntu on the phone, and the protocol implementation was compiled to AArch64 native binaries.

The main server was the 5G MEC application server, with the following resources:

- 16 GB memory
- 2 Virtual CPU cores

¹https://www.gsmarena.com/xiaomi_poco_x6-12723.php

²<https://github.com/CypherpunkArmory/UserLAnd>

The server used for NSA measurements:

- 16 GB memory
- Intel Celeron G6900

Every test was compiled with release-mode³ Rust, benchmarks were executed as unit tests, and the server was executed as a regular process. All processes were (re)started for each test to avoid state or counters skewing the result.

The software used by the client is as follows:

- Rust 1.81
- Tokio 1.40
- Ubuntu 24.04 on clients
- Ubuntu 22.04 LTS on servers

Devices running Windows were also used both as client and server to detect platform-specific behaviour, but for the sake of consistency, only Linux measurements are shown. The performance differences between platforms were negligible.

Measurements that use the *NWEmg* protocol use DTLS as per the specification of the protocol, even in *measure* mode.

In tests where message size is not specified explicitly, messages of 168 64-bit floats are used (the payload is 1344 bytes in total) as by the time messages are encapsulated in NWEmg datagrams, go through DTLS encryption, are sent over UDP and are encapsulated in a layer 2 frame they come close to the minimal (1456 byte) MTU measured across all test environments. These messages are the largest sent throughout all benchmarks and are 1424 bytes long (with Ethernet encapsulation, measured in Wireshark⁴).

4.1.2 The UDPEcho Utility

A purely synchronous UDP client and server pair was created to verify or disprove theories surrounding packet ordering, queuing and the impact of DTLS. The *NWEmg* protocol relies on asynchronous tasks, therefore a purely synchronous application that serves as the control in each networking environment is desirable, as it helps with the detection and identification of the cause of reordering, be it a network segment or an application module.

The tool helped me discover the extreme impact of queuing in scheduled networks that in the most extreme cases resulted in the asynchronous code dropping 60-70% of the messages due to a newer message being received by the time the messages pass DTLS decryption.

4.1.3 Further Validation Using IPerf and ICMP Messages

To validate the results measured using my custom tools, I ran tests with IPerf and used the Linux ping command with parameters set as close to the protocol's parameters as possible. With both utilities, I managed to produce results within an acceptable margin of error.

³<https://doc.rust-lang.org/cargo/reference/profiles.html#release>

⁴<https://www.wireshark.org/>

4.1.4 The Measure Module

To simplify benchmarking the protocol, a module with this specific purpose was developed along with the rest of the native code base. The module collects all results in in-memory data structures (in the current implementation a pre-allocated vector with a capacity of 100 000 log items) in order to minimize the impact of using the standard output. At the end of each session, a CSV version of these structures is written into a timestamped log file, prefixed with the name of the test performed.

The module provides various benchmarking features with a configurable message size, frequency, and data source (e.g., randomly generated or imported real-world data).

4.1.5 Graphs and Tables

For all tests, the same set of graphs and tables were generated using a tool I created to speed up the analysis of the data measured using Python, Matplotlib, Seaborn, and Pandas.

I chose to create graphs showing the roundtrip time of each message by its rank, scatter-plots that show RTT throughout the session and ECDF plots that show the distribution of the RTTs of messages.

In each case, top 99%, top 95% and top 90% versions of the graphs were created. The most readable version is shown here. Every diagram generated for each test case is available in the Appendix.

For every scenario, two tables are provided. The first table lists information on RTT averages, the columns are:

- mean RTT: The mean RTT of all messages
- p95: The 95th percentile RTT
- p95 mean: The mean RTT of the top 95% of all messages
- p90: The 90th percentile RTT
- p90 mean: The mean RTT of the top 90% of all messages
- median: The median RTT

The second table lists statistics about message delivery, specifically

- max. pending: The maximum interval for which the client received no updates
- non-unique: the number of messages the protocol registered while an earlier message was still expected. Usually indicates datagram loss or out-of-order/grouped reception.
- mean and maximum jitter
- timed out: the number of replies that received no replies within 500 ms

Note that the axes of tables are scaled to fit the data presented, and are not uniform across sections. To make comparisons easier, a dedicated section (Comparison of Technologies) shows compared data.

RTT values are shown in seconds.

4.2 Results

Each measurement presented in the following subsections shows the results of sessions for each parameter set with 100,000 messages sent. All networks provided reliable enough connections that none of the sessions was terminated due to errors.

4.2.0.1 Baseline

This section shows the results of the protocol running on localhost, and through Ethernet, as Ethernet was used to connect the PC to the 5G CPEs.

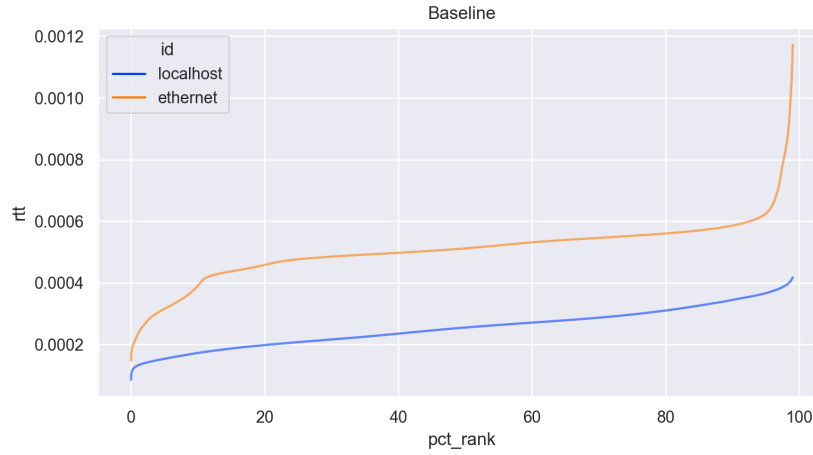


Figure 4.1: Baseline Rank to RTT (top 99%)

The protocol has an about 0.3 ms average processing time, and Ethernet has an average overhead of about 0.5 ms including the 0.3 ms processing.

4.2.1 5G Benchmarks

4.2.1.1 5G SA

This subsection shows measurements carried out on the BME-Nokia 5G research network⁵. The research network uses Nokia DAC⁶ private 5G devices, the network is one of the largest private academic networks in the EU. During all measurements, the CPE had a clear line of sight to the pico remote radio unit (pRRU). The path MTU was 1456 bytes.

Note that 5G SA has low max. pending values across all message rates with no messages reaching the 500 ms timeout.

The measurements were conducted using a NOKIA FastMile 5G Gateway 3.2 CPE⁷ (5G15-12W-A), and the results were verified using a Hocell 5G CPE M110⁸. The PC conducting the measurements was connected to the CPE through Ethernet. Concerning the latency introduced by Ethernet, see the results in Baseline.

⁵<https://5g.bme.hu/>

⁶<https://www.dac.nokia.com/>

⁷<https://www.nokia.com/networks/fixed-networks/fastmile/5g-gateway-32-2/>

⁸<https://www.hocell.com/productinfo/535021.html>

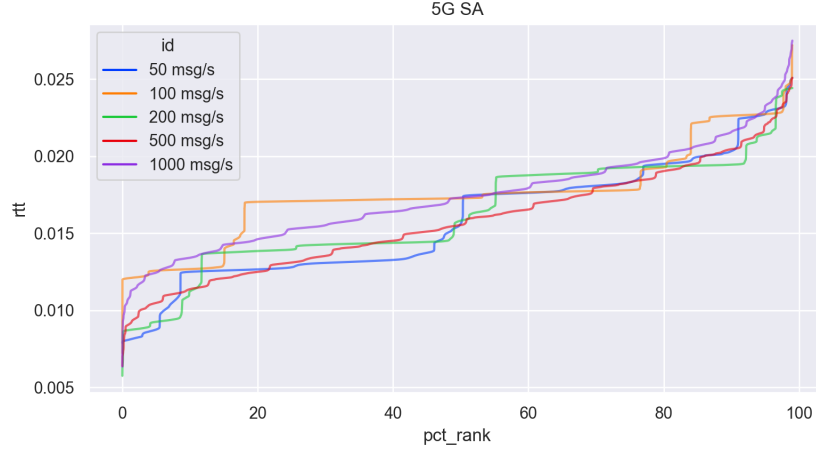


Figure 4.2: 5G SA Rank to RTT (top 99%)

The scatterplot shows distinct bands of RTTs, most likely due to scheduling, mostly affecting uplink, as the server receives a significant subset of received messages in groups of two or in some cases even more.

Interestingly, going from 200 messages to 500 messages per second resulted in the messages being treated completely differently, most likely due to the characteristics of the message flow getting closer to more common patterns like voice traffic.

5G NR had no problem handling the increase in message rate, and the effects of scheduling are visible on the ECDF diagram too. The latency measured is comparable to measurements of similar systems [8] and [26].

	mean rtt	p95	p95 mean	p90	p90 mean	median
50 msg/s	0.0159	0.0227	0.0155	0.0203	0.0151	0.0156
100 msg/s	0.0176	0.0227	0.0173	0.0226	0.0170	0.0173
200 msg/s	0.0163	0.0214	0.0159	0.0194	0.0156	0.0158
500 msg/s	0.0159	0.0220	0.0154	0.0205	0.0151	0.0155
1000 msg/s	0.0175	0.0233	0.0170	0.0216	0.0167	0.0173

Table 4.1: 5G SA RTT statistics

	max. pending	non-unique	mean jitter	max. jitter	timed out
50 msg/s	0.0856	24	0.0054	0.0656	0
100 msg/s	0.0331	11431	0.0033	0.0231	0
200 msg/s	0.0298	12185	0.0048	0.0248	0
500 msg/s	0.0259	9474	0.0030	0.0239	0
1000 msg/s	0.0230	10819	0.0016	0.0220	0

Table 4.2: 5G SA Delivery statistics

4.2.1.2 5G NSA (Vodafone)

A second batch of measurements was conducted on the commercial 5G network using a 5G capable phone listed in Devices and software), and a non-MEC server 12 hops away from the UE.

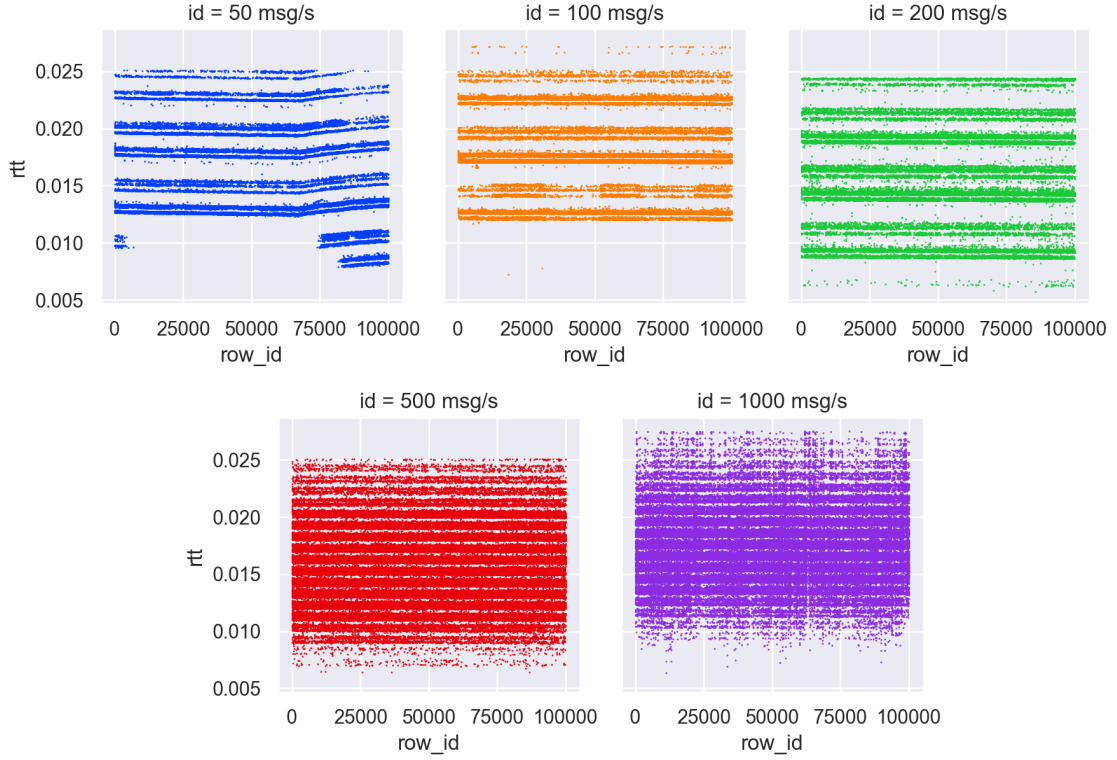


Figure 4.3: 5G SA scatterplots (top 99%)

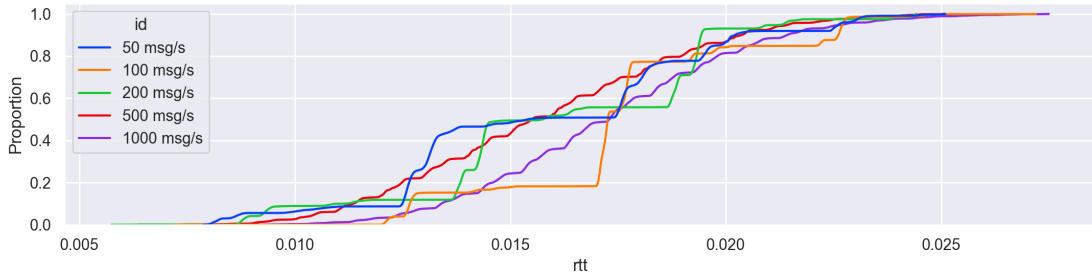


Figure 4.4: 5G SA ECDF (top 99%)

I decided to benchmark 5G NSA as most carriers provide only (or at least mostly) non-standalone 5G, and the adaptation of SA 5G seems stagnant⁹.

The first 5 sets of data shown in this section are the results of measurements taken indoors in a residential house. The last dataset is the result of a measurement taken during a 500 m walk in the area covered by the same cell used by the first 5 measurements.

The cell I used for measurements was an NR cell (ECI 1565452), that offers NSA 5G (reported as NR State (NSA): connected¹⁰). The path MTU was 1492 bytes.

While the results are considerably worse than those measured on 5G SA, considering the 150 ms initial goal, the results still leave plenty of time for processing.

⁹<https://www.delloro.com/news/mobile-core-market-stagnant-due-to-lack-of-5g-standalone-deployments/>

¹⁰<https://source.android.com/docs/core/connect/5g-nsa>

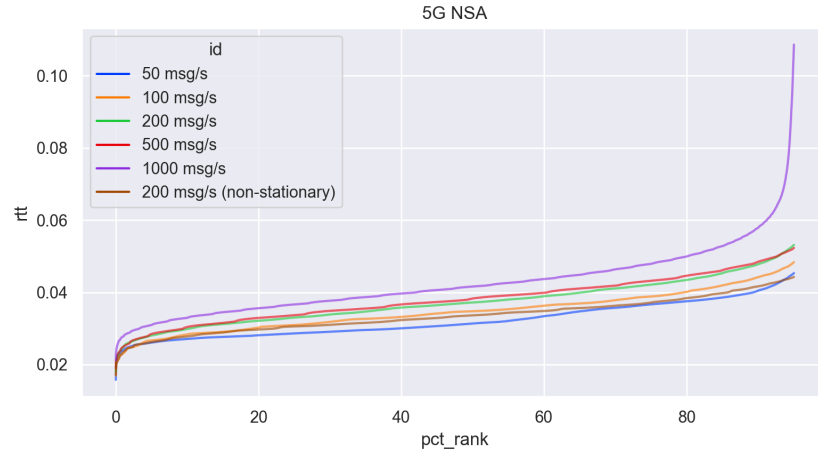


Figure 4.5: 5G NSA Rank to RTT (top 95%)

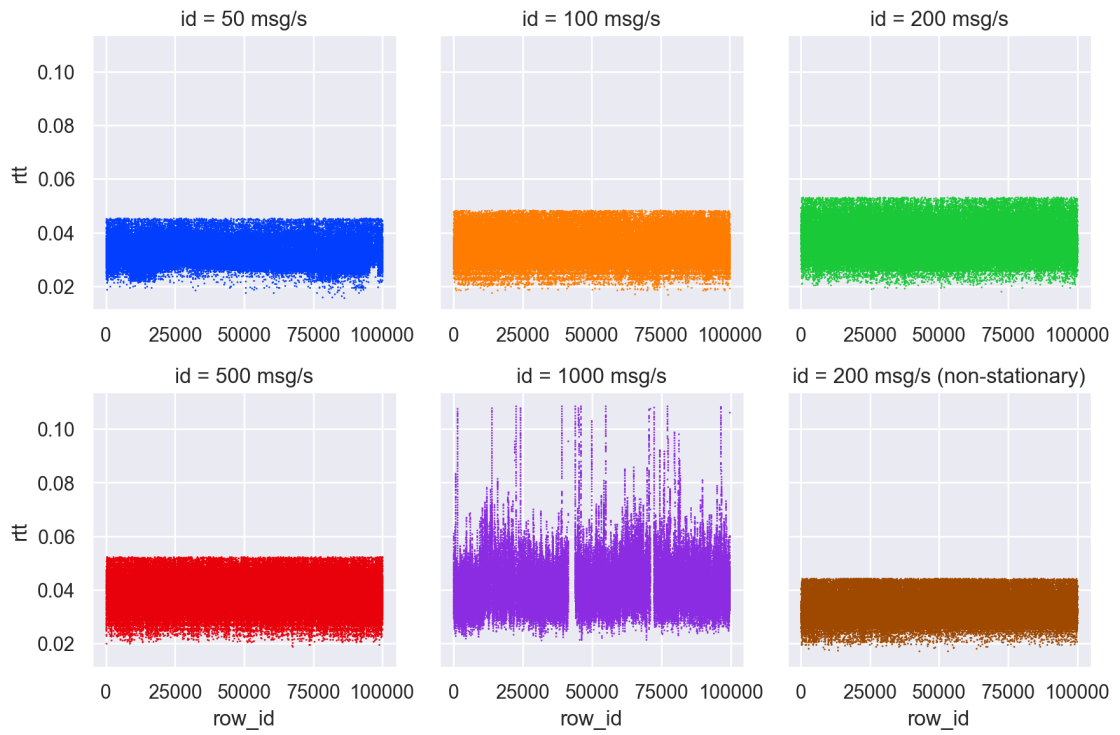


Figure 4.6: 5G NSA scatterplots (top 95%)

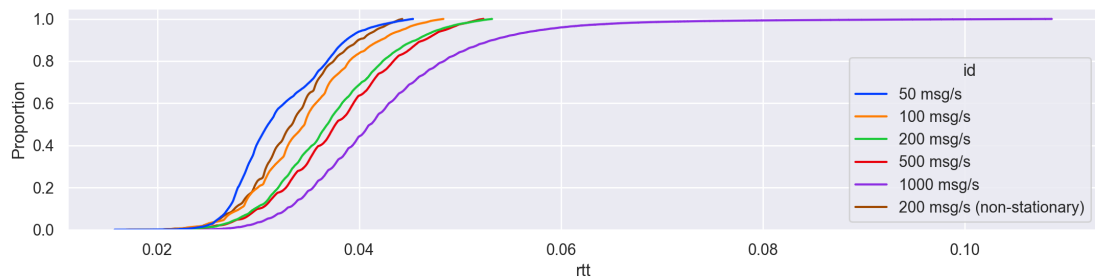


Figure 4.7: 5G NSA ECDF (top 95%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
50 msg/s	0.0356	0.0453	0.0322	0.0405	0.0316	0.0314
100 msg/s	0.0363	0.0483	0.0346	0.0442	0.0340	0.0348
200 msg/s	0.0397	0.0532	0.0372	0.0478	0.0365	0.0372
500 msg/s	0.0397	0.0523	0.0380	0.0485	0.0373	0.0383
1000 msg/s	0.0570	0.1087	0.0425	0.0579	0.0411	0.0416
200 msg/s ^{moving}	0.0344	0.0443	0.0335	0.0417	0.0330	0.0336

Table 4.3: 5G NSA RTT statistics

	max. pending	non-unique	mean jitter	max. jitter	timed out
50 msg/s	0.4914	2222	0.0069	0.4714	98
100 msg/s	0.2692	7854	0.0056	0.2592	0
200 msg/s	0.1497	15835	0.0047	0.1414	4
500 msg/s	0.1313	14515	0.0030	0.1293	0
1000 msg/s	0.1281	16146	0.0017	0.1241	1343
200 msg/s ^{moving}	0.1058	14581	0.0039	0.0987	0

Table 4.4: 5G NSA delivery statistics

4.2.2 Wi-Fi

Both Wi-Fi 5 and 6 were benchmarked because Wi-Fi 6 brought such a substantial improvement to the 802.11 standard family that I feel like Wi-Fi 6 might be a viable alternative to 5G NR inside buildings, while Wi-Fi 5 could not have been. Wi-Fi 6 introduced OFDMA [18] and is available even in some low-cost microcontrollers (such as the ESP32-C6). Measurements on both Wi-Fi standards yielded surprising results, the difference is more apparent in section Comparing all technologies. The devices were in the same room as the Wi-Fi access points (APs) they were connected to.

4.2.2.1 Wi-Fi 5

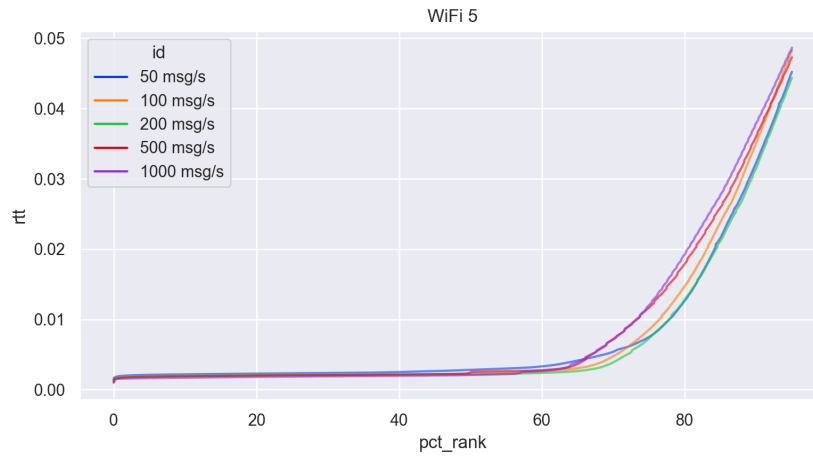


Figure 4.8: Wi-Fi 5 Rank to RTT (top 95%)

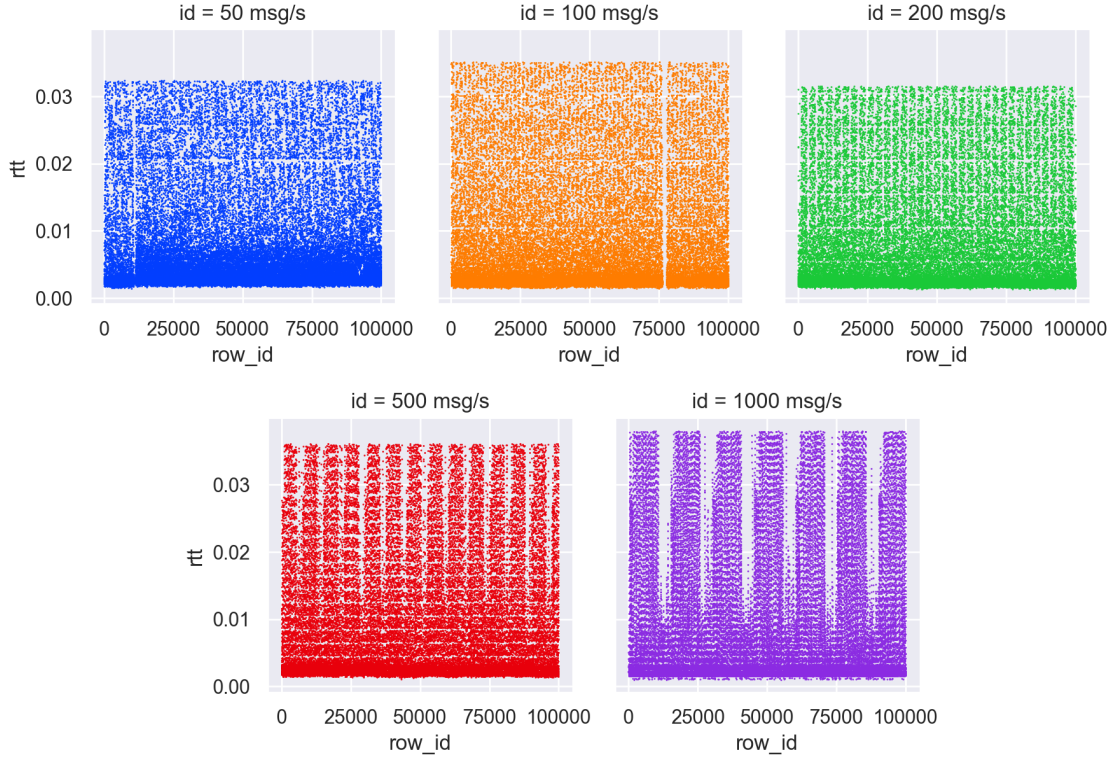


Figure 4.9: Wi-Fi 5 scatterplots (top 90%)

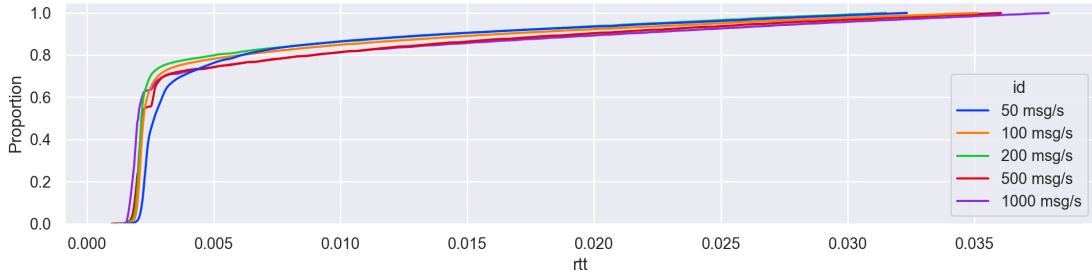


Figure 4.10: Wi-Fi 5 ECDF (top 90%)

I decided to start with the older Wi-Fi 5 results, as they are surprisingly good at first glance. Wi-Fi 5 prompted the generation of top 90% diagrams, as the worst 10% was exceptionally bad compared to the top 60%, making it the most inconsistent of all technologies benchmarked. The longest period without an update was also measured on Wi-Fi 5. Wi-Fi 5 had the longest pending time, the time for which the client had not received an update.

	mean rtt	p95	p95 mean	p90	p90 mean	median
50 msg/s	0.0124	0.0452	0.0072	0.0323	0.0054	0.0028
100 msg/s	0.0169	0.0483	0.0074	0.0351	0.0055	0.0023
200 msg/s	0.0090	0.0444	0.0067	0.0315	0.0049	0.0022
500 msg/s	0.0104	0.0473	0.0080	0.0360	0.0061	0.0025
1000 msg/s	0.0106	0.0487	0.0082	0.0379	0.0063	0.0021

Table 4.5: Wi-Fi 5 RTT statistics

	max. pending	non-unique	mean jitter	max. jitter	timed out
50 msg/s	0.5189	4512	0.0086	0.4989	649
100 msg/s	0.5809	4612	0.0053	0.5609	1493
200 msg/s	0.1336	4383	0.0030	0.1286	0
500 msg/s	0.1308	6416	0.0016	0.1288	0
1000 msg/s	0.1072	7916	0.0009	0.1062	0

Table 4.6: Wi-Fi 5 delivery statistics

4.2.2.2 Wi-Fi 6

Wi-Fi 6 provided consistency comparable to 5G NR while achieving much lower average latency. The increased latency with 1000 messages/s was reproducible using another device.

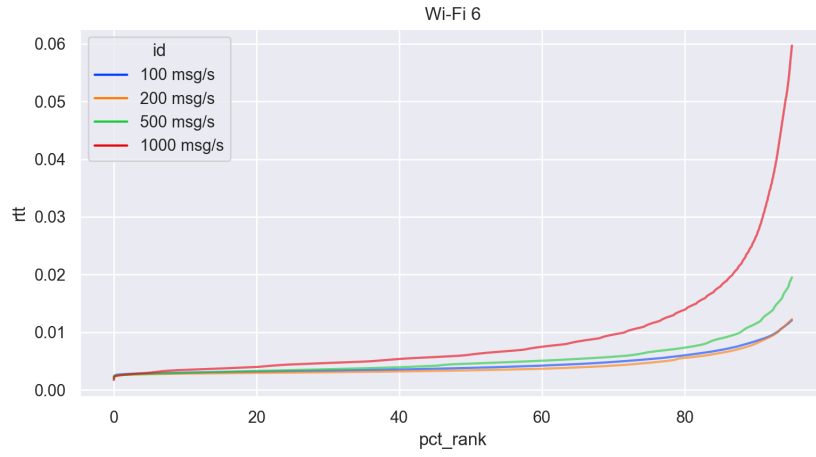


Figure 4.11: Wi-Fi 6 Rank to RTT (top 95%)

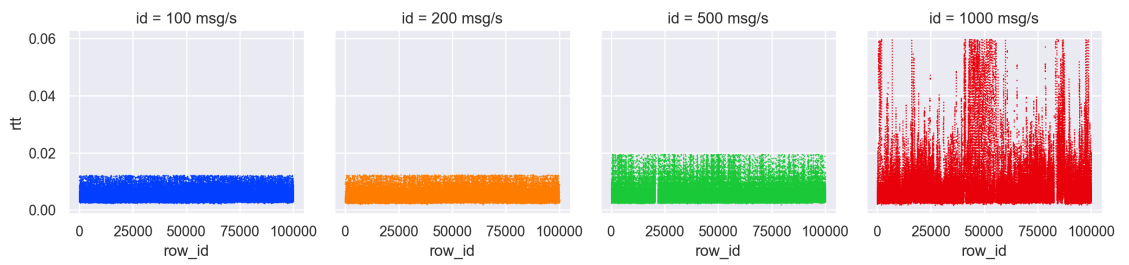


Figure 4.12: Wi-Fi 6 scatterplots (top 95%)

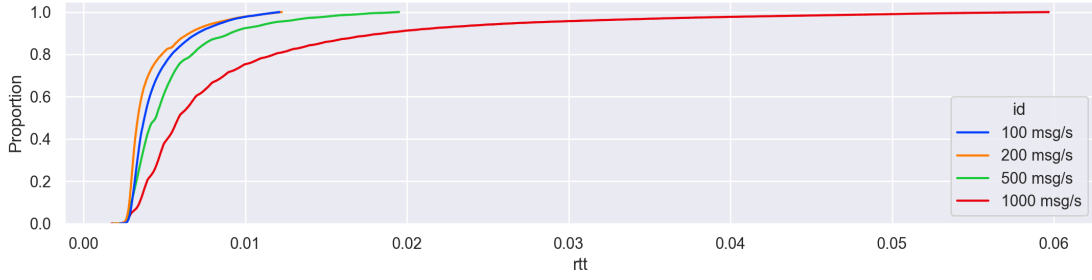


Figure 4.13: Wi-Fi 6 ECDF (top 95%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
100 msg/s	0.0056	0.0121	0.0044	0.0085	0.0041	0.0038
200 msg/s	0.0055	0.0123	0.0041	0.0081	0.0038	0.0034
500 msg/s	0.0098	0.0195	0.0053	0.0115	0.0048	0.0046
1000 msg/s	0.0163	0.0597	0.0092	0.0267	0.0075	0.0061

Table 4.7: Wi-Fi 6 RTT statistics

	max. pending	non-unique	mean jitter	max. jitter	timed out
100 msg/s	0.4198	1075	0.0025	0.3798	0
200 msg/s	0.1654	2575	0.0018	0.1604	0
500 msg/s	0.1317	8479	0.0017	0.1263	192
1000 msg/s	0.2476	15168	0.0015	0.2468	134

Table 4.8: Wi-Fi 6 Delivery statistics

4.2.3 Measurements Besides the Shown Results

Several other measurements were conducted to ensure that the benchmarks shown above are not specific to the devices they were executed on and are reproducible in different environments. In addition to validating the measurements using different software, the 5G NR measurements were compared with control measurements using a different small cell in BME building I. WiFi measurements which used a phone as the client were executed both in BME building Q and I, and measurements were also compared to control measurements I conducted using an 802.11ax capable laptop.

Measurements with a Server-side Executor classifying actual data sent by the client (using a test dataset) were also conducted. As expected, the results differ only in the added processing overhead of the classifier model.

Benchmarks using messages with different payload sizes are included in the Appendix.

4.2.4 Comparison of Technologies

Even though the technologies shown are significantly different in their characteristics, a side-by-side comparison cannot be left out. 200 messages/s seemed to be the optimal spot for most technologies and it is also the expected message rate in a real-life setting. Surprisingly, no technology did significantly worse at 200 messages/s than at lower rates, and in most cases, lower than 200 messages/s message rates yielded worse results.

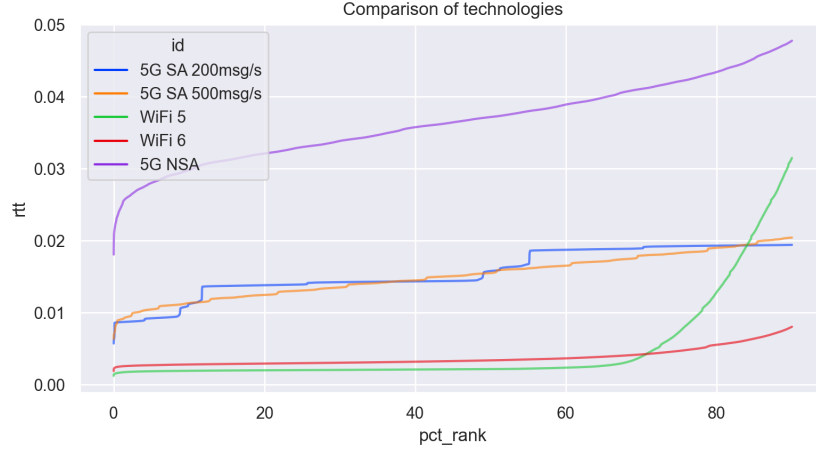


Figure 4.14: All technologies compared (Rank to RTT, top 90%)

The comparison shows the top 90% data so that differences beyond Wi-Fi 5 being inconsistent are visible. All wireless technologies I benchmarked handled the measurements without losing connection, and even the worst results are good enough that I feel confident that under ideal conditions all access technologies are capable of handling the task.

The non-unique value in some cases is surprisingly high. To reduce the number of non-unique registrations a client-side buffer that attempts to restore the correct (sequential) order within a given time limit could be used, nevertheless, in this use case, it does not make sense to wait for an earlier message once a newer one is available.

	mean rtt	p95	p95 mean	p90	p90 mean	median
5G SA ^{200msg/s}	0.0163	0.0214	0.0159	0.0194	0.0156	0.0158
5G SA ^{500msg/s}	0.0159	0.0220	0.0154	0.0205	0.0151	0.0155
WiFi 5	0.0090	0.0444	0.0067	0.0315	0.0049	0.0022
WiFi 6	0.0055	0.0123	0.0041	0.0081	0.0038	0.0034
5G (NSA)	0.0397	0.0532	0.0372	0.0478	0.0365	0.0372

Table 4.9: RTT statistics (comparison of technologies)

	max. pending	non-unique	mean jitter	max. jitter	timed out
5G SA ^{200msg/s}	0.0298	12185	0.0048	0.0248	0
5G SA ^{500msg/s}	0.0259	9474	0.0030	0.0239	0
WiFi 5	0.1336	4383	0.0030	0.1286	0
WiFi 6	0.1654	2575	0.0018	0.1604	0
5G NSA	0.1497	15835	0.0047	0.1414	4

Table 4.10: Delivery statistics (comparison of technologies)

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Intelligent prostheses that offload the processing of EMG data rather than processing measurements locally could provide a better user experience, as algorithms executed by edge cloud solutions are far less constrained in complexity and energy requirements.

I designed a protocol and created a framework that facilitates the process of offloading EMG processing. The framework also provides benchmarking utilities that can help determine the viability of offloaded processing in certain networking environments. The framework was implemented in Rust, but the server and client components are interoperable with Python and provide an environment which makes the development and testing of server-side models faster. The client also exposes most of its native functionality to Python through bindings and provides a simple yet performant API for testing and validation.

While most carriers are yet to deploy 5G MEC servers in their networks, most multinational carriers already market MEC capabilities as part of their offerings, and the technology is the subject of many active research projects. The demand for MEC deployments should increase substantially with the appearance of new applications that directly benefit from predictable (low) latency or high bandwidth.

I conducted extensive benchmarks and experiments on several networks in various contexts that helped us better understand the characteristics and behaviour of different wireless networking technologies and edge cloud solutions. Based on these benchmarks, we can conclude that modern wireless networking technologies in combination with lightweight domain-specific protocols are capable of transmitting data reliably enough to be suitable for offloading control-related computations in situations, where an end-to-end latency of N or at most $N \cdot 10$ ms is tolerable.

The protocol I created is a minimal-overhead, relatively simple solution for handling low-volume, time-critical and domain-specific communication complete with EMG processing.

The results presented in this paper are universally applicable to other domains with similar latency and bandwidth requirements. Even in less time-critical applications benefits might come with offloading computations, the most prominent being the availability of significantly more powerful computational resources with virtually no energy constraints.

With the incorporation of 5G and B5G technologies, the protocol could handle network errors or changes in network parameters seamlessly, and with the use of QoS flows, the bandwidth required to deliver messages could be guaranteed.

Our tests showed that Wi-Fi 6 (along with later versions of Wi-Fi) might be a viable alternative to 5G NR inside buildings where 5G has worse coverage, and in areas where 5G coverage is not yet available but the access to MEC hosts is feasible. 5G provided profound consistency and range during our measurements and is advantageous as it has proper support for mobility such as reliable handover mechanisms.

The MEC orchestrator can ensure that the application is deployed on a MEC host that provides the least latency on a per-user basis, and ensures proper scaling if required.

The goals we have set in Goals were not only met but according to the measurements, a significant part of the 150 ms e2e latency remains available for server-side processing.

Further research is required to uncover the exact accuracy gain this solution could provide, and whether the data collected during normal operations could be subjected to offline analysis to further improve efficiency.

Ultimately, controlling intelligent prostheses can be and hopefully will be one of the prime use cases for 5G, B5G and edge cloud services.

5.2 Future Work

The project has reached the stage in which the next steps are the deployment of the solution in an experimental physical setting (i.e. on a physical prosthesis connected to the MEC application through 5G or Wi-Fi) and benchmarking and testing the accuracy and performance of different classifier models be executed by the server.

5.2.1 Guaranteed Transmission

The implementation of the *DataGuaranteed* message type that uses network statistics to determine the need for redundancy or simple retransmission is yet to be finalized/tested, as for benchmarking networks, and undergoing tests (even for those that use models) no guaranteed transmission was required.

5.2.2 Analysis of Historical Data

The current implementation of the server retains the data received for logging and analysis. Further research is required to determine whether the data collected could be used to increase accuracy.

5.2.3 Parallel Execution of Models

There are different models available for EMG signal classification, and different models might be prone to different causes of incorrect classifications. The MEC application might have enough resources to execute multiple models in parallel, the fusion of results could be used as the result. It requires further research whether such a solution is capable of improving efficiency.

5.2.4 IPv6 and Path MTU Discovery

The protocol was designed to be used with payload sizes below the path MTU. As per section 3.2 of RFC 8085 [11] an application should not send UDP datagrams that result in fragmentation. Should the need arise to support larger than MTU-sized messages the protocol needs to be extended with application-layer fragmentation (and path-MTU detection, especially over IPv6, IPv4 routers will fragment packets at the cost of reliability and efficiency). IPv6 increases the minimum MTU to 1280 bytes[10] which might make the need for fragmentation slightly less likely.

5.2.5 Custom Payloads

To make control messages more flexible, a fixed-size byte array similar to the error string in *ErrorReport* messages might be introduced for the transmission of custom binary data as in some scenarios *server-side executors* might benefit from allowing custom parameters to be received from clients.

5.2.6 5G and Beyond 5G Technologies

Radio network information service (RNIS) [2] (also available for Wi-Fi [3] and is under active development [9]) could be used to reconfigure the parameters of the data transmission to maximize efficiency or compensate for lossy transmission using redundancy, or make the decision to use the locally available fallback model until conditions are normalized.

QoS flows[5]: As the bandwidth requirements of the protocol are well-predictable (disregarding control messages it is declared upon initialization), a guaranteed bitrate (GBR) QoS flow could ensure the reliable delivery of EMG data.

N3IWF [4]: The 5G specification provides a standardized way of incorporating access to the 5G core network (and as such, MEC) through non-3GPP access networks. As outlined in GSMA's requirement specification for VoWi-Fi [14], Wi-Fi networks have the potential to provide better indoor coverage, and accessing the MEC application running in a MEC host that is part of a 5G network through Wi-Fi is possible.¹

5.2.7 Integration With a Physical Prosthesis

After further testing of the protocol while running algorithms, the protocol will be deployed to a physical prosthesis. During the first round of testing, the client will run on an external device, but later a 5G capable embedded client might be used.

5.2.8 Embedded Targets

The client has no problem running on a low-power system such as a Raspberry Pi 3, Rust can target embedded "Bare Metal" devices², which might be the final target for the client library. In its final form, the protocol might publish a drop-in library with API that provides a user-friendly solution for low-bandwidth near-real-time communication (similarly to how Arduino supports UART through the `Serial`³ class).

¹<https://www.linkedin.com/pulse/n3iwf-mec-gpon-5g-abu-hayat-khan/>

²<https://docs.rust-embedded.org/book/>

³<https://www.arduino.cc/reference/tr/language/functions/communication/serial/>

5.2.9 Deployment and Scaling

The deployment and orchestration of MEC applications are out of this project's scope, but the protocol was designed with simultaneous sessions in mind, and the task pool processing the requests will scale well as long as there are available host resources. For now, it is assumed that the application is orchestrated with a one instance per user model out of the possibilities specified in the standard connected to MEC orchestration [1]. This model might be wasteful of host resources but ensures that all resources of a dedicated instance are available to each user and makes migration from one host to another less complicated. Research into resource allocation [23] and energy optimization of MEC applications [21] is ongoing. Tests in connection with optimal resource usage and the impact of multiple users and vertical scaling are part of my plans.

Acknowledgement

This research was supported by the Ministry of Culture and Innovation and the National Research, Development and Innovation Office within the Cooperative Technologies National Laboratory of Hungary (grant No. 2022-2.1.1-NL-2022-00012).

Bibliography

- [1] ETSI GS MEC 003. Multi-access edge computing (mec); framework and reference architecture. Technical specification (ts), 2022. URL http://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/03.01.01_60/gs_MEC003v030101p.pdf.
- [2] ETSI GS MEC 012. Multi-access edge computing (mec); radio network information api. Technical specification (ts), 2019. URL https://www.etsi.org/deliver/etsi_gs/MEC/001_099/012/02.01.01_60/gs_mec012v020101p.pdf.
- [3] ETSI GS MEC 028. Multi-access edge computing (mec); wlan access information api. Technical specification (ts), 2021. URL https://www.etsi.org/deliver/etsi_gs/MEC/001_099/028/02.02.01_60/gs_MEC028v020201p.pdf.
- [4] 3GPP. Access to the 3GPP 5G Core Network (5GCN) via non-3GPP access networks. Technical Specification (TS) 24.502, 3rd Generation Partnership Project (3GPP), . URL <http://www.3gpp.org/DynaReport/24502.htm>.
- [5] 3GPP. 5G System; Policy and Charging Control signalling flows and QoS parameter mapping; Stage 3. Technical Specification (TS) 29.513, 3rd Generation Partnership Project (3GPP), . URL <http://www.3gpp.org/DynaReport/29513.htm>.
- [6] ETSI White Paper No. 55. Mec support towards edge native design. Technical report, 2023. URL https://www.etsi.org/images/files/ETSIWhitePapers/ETSI-WP55-EC_support_towards_Edge_native.pdf.
- [7] Adnan Aijaz. Private 5g: The future of industrial wireless. *IEEE Industrial Electronics Magazine*, 14(4):136–145, 2020. DOI: 10.1109/MIE.2020.3004975.
- [8] Sachinkumar Bavikatti Mallikarjun, Christian Schellenberger, Christopher Hobelsberger, and Hans Schotten. Performance analysis of a private 5g sa campus network. 02 2023.
- [9] Kaíque Cunha, Sand Correa, Fabrizzio Soares, Maria Ribeiro, Waldir Moreira, Raphael Gomes, Leandro Freitas, and Antonio Oliveira Junior. A novel radio network information service (rnis) to mec framework in b5g networks. *Information*, 15:352, 06 2024. DOI: 10.3390/info15060352.
- [10] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017. URL <https://www.rfc-editor.org/info/rfc8200>.
- [11] Lars Eggert, Gorrry Fairhurst, and Greg Shepherd. UDP Usage Guidelines. RFC 8085, March 2017. URL <https://www.rfc-editor.org/info/rfc8085>.
- [12] Abderrahime Filali, Amine Abouaoumar, Soumaya Cherkaoui, Abdellatif Kobbane, and Mohsen Guizani. Multi-access edge computing: A survey. *IEEE Access*, 8: 197017–197046, 2020. DOI: 10.1109/ACCESS.2020.3034136.

- [13] Moinak Ghoshal, Pranab Dash, Z. Jonny Kong, Qiang Xu, Y. Hu, Dimitrios Koutsounikolas, and Yuanjie Li. *Can 5G mmWave Support Multi-user AR?*, pages 180–196. 01 2022. ISBN 978-3-030-98784-8. DOI: 10.1007/978-3-030-98785-5_8. URL <https://repository.library.neu.edu/files/neu:4f17qg40k/fulltext.pdf>.
- [14] GSMA. Ts.63 5g sa ue wi-fi calling requirements specification. 2024. URL https://www.gsma.com/get-involved/working-groups/gsma_resources/ts-63-v1-0-ue-wi-fi-calling-requirements-specification/.
- [15] Malleswar Kalla, Randall R. Stewart, Tom Taylor, Dr. Vern Paxson, Chip Sharp, Ken Morneault, Dr. HannsJuerge Schwarzbauer, Qiaobing Xie, Ian Rytina, and Lixia Zhang. Stream Control Transmission Protocol. RFC 2960, October 2000. URL <https://www.rfc-editor.org/info/rfc2960>.
- [16] Jorge LEÓN-PERALTA, R. Sanchez-Lara, J.L. Vazquez-Avila, and Juan YAÑEZ-VARGAS. Implementation of a neural network of low computational cost for its application in arm prostheses. *Revista de Ingeniería Tecnológica*, pages 27–34, 11 2022. DOI: 10.35429/JTEN.2022.17.6.27.34.
- [17] Yaqiong Liu, Mugen Peng, Guochu Shou, Yudong Chen, and Siyu Chen. Toward edge intelligence: Multiaccess edge computing for 5g and internet of things. *IEEE Internet of Things Journal*, 7:6722 – 6747, 06 2020. DOI: 10.1109/JIOT.2020.3004500.
- [18] Erfan Mozaffariahrar, Fabrice Theoleyre, and Michael Menth. A survey of wi-fi 6: Technologies, advances, and challenges. *Future Internet*, 14:293, 10 2022. DOI: 10.3390/fi14100293.
- [19] Gianfranco Nencioni, Rosario Garroppo, and Ruxandra Olimid. 5g multi-access edge computing: A survey on security, dependability, and performance. *IEEE Access*, PP: 1–1, 01 2023. DOI: 10.1109/ACCESS.2023.3288334.
- [20] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, January 2012. URL <https://www.rfc-editor.org/info/rfc6347>.
- [21] Rasha Samir, Hadia El-Hennawy, and Hesham M. El-Badawy. Orchestration of mec computation jobs and energy consumption challenges in 5g and beyond. *IEEE Access*, 10:18645–18652, 2022. DOI: 10.1109/ACCESS.2022.3151389.
- [22] Oluwarotimi Samuel, Mojisola Asogbon, Yanjuan Geng, Ali Al-Timemy, Sandeep Pirbhulal, Ning Ji, Shixiong Chen, Peng Fang, and Patrick Li. Intelligent emg pattern recognition control method for upper-limb multifunctional prostheses: Advances, current challenges, and future prospects. *IEEE Access*, PP:1–1, 01 2019. DOI: 10.1109/ACCESS.2019.2891350.
- [23] Annisa Sarah, Gianfranco Nencioni, and Md. Muhidul I. Khan. Resource allocation in multi-access edge computing for 5g-and-beyond networks. *Computer Networks*, 227:109720, 2023. ISSN 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2023.109720>. URL <https://www.sciencedirect.com/science/article/pii/S1389128623001652>.
- [24] Michael Scharf and Sebastian Kiesel. Head-of-line blocking in tcp and sctp: Analysis and measurements. pages 1 – 5, 01 2007. DOI: 10.1109/GLOCOM.2006.333. URL <https://ieeexplore.ieee.org/abstract/document/4150963>.

- [25] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014. URL <https://www.rfc-editor.org/info/rfc7252>.
- [26] Nunzio Torrisi, José Meira, Gonçalo Matos, André Perdigão, José Cação, Carlos Resende, Waldir Moreira, Mário Antunes, José Quevedo, Ruben Moutinho, João Oliveira, Pedro Rendeiro, Pedro Oliveira, Antonio Oliveira Junior, José Santos, and Rui Aguiar. Industrial internet of things over 5g: A practical implementation. *Sensors*, 23, 05 2023. DOI: 10.3390/s23115199. URL https://www.researchgate.net/publication/371169317_Industrial_Internet_of_Things_over_5G_A_Practical_Implementation.
- [27] Nicolas Witt, Mark Deutel, Jakob Schubert, Christopher Sobel, and Philipp Woller. *Energy-Efficient AI on the Edge*, pages 359–380. 07 2024. ISBN 978-3-031-64831-1. DOI: 10.1007/978-3-031-64832-8_19. URL https://www.researchgate.net/publication/382679690_Energy-Efficient_AI_on_the_Edge.

Appendix

The appendix contains all diagrams generated for each measurement.

A.1 5G SA

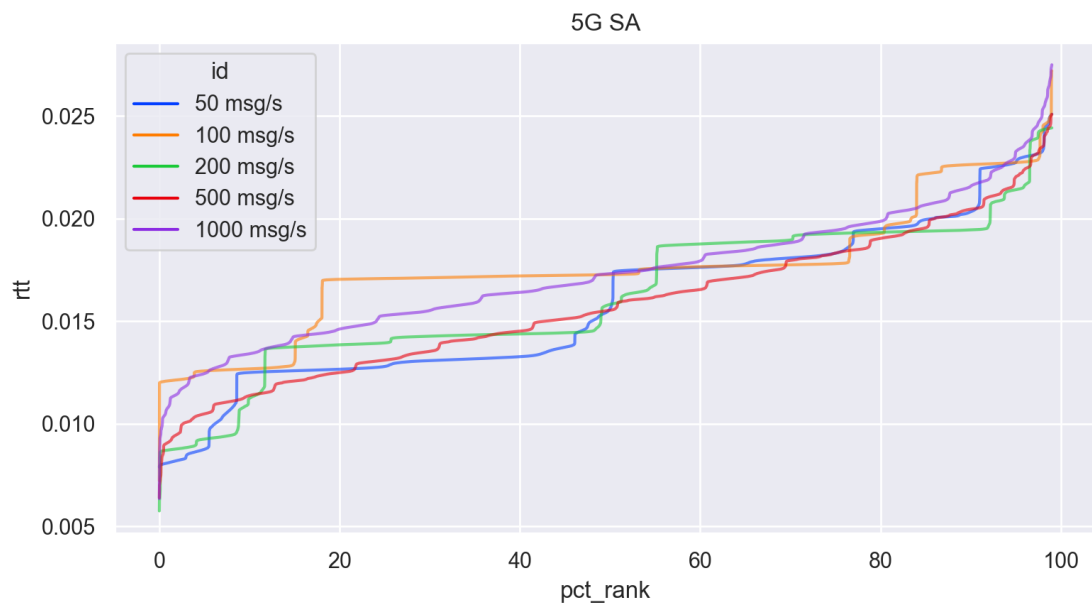


Figure A.1.1: 5G SA (Rank to RTT, top 99%)

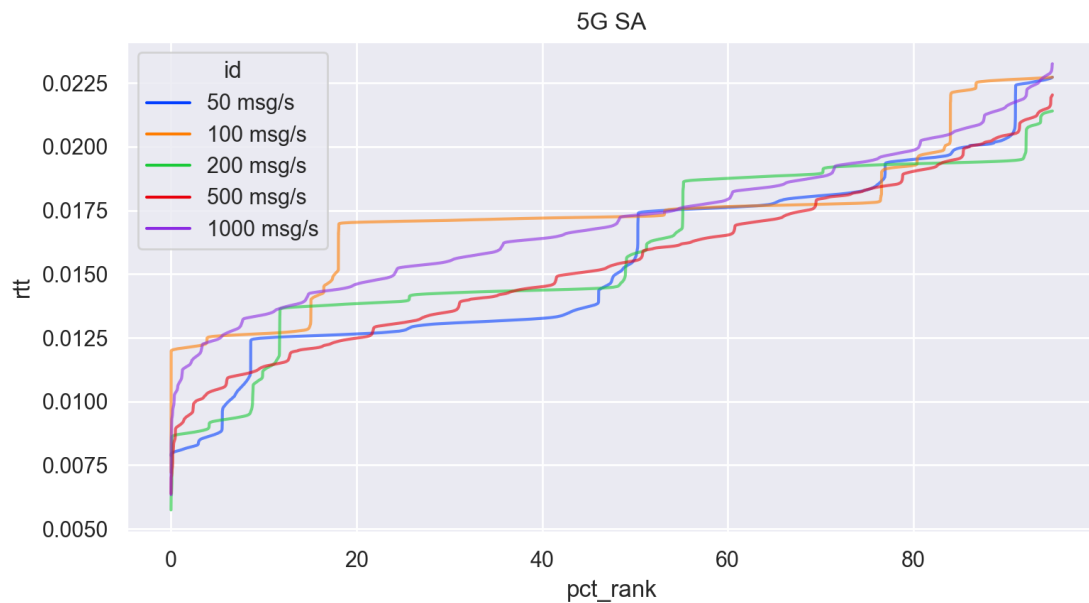


Figure A.1.2: 5G SA (Rank to RTT, top 95%)

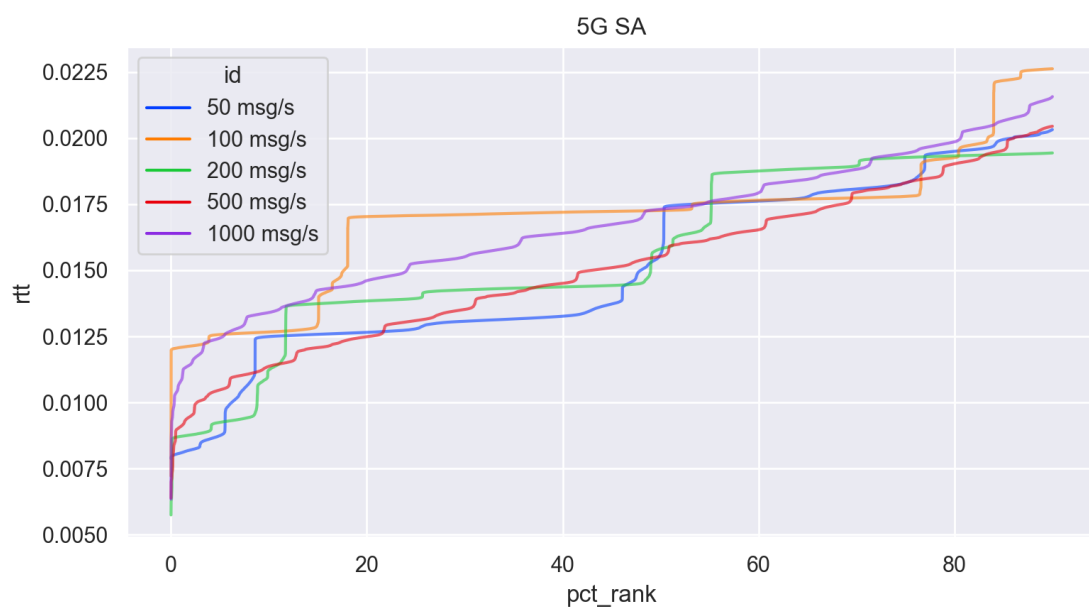


Figure A.1.3: 5G SA (Rank to RTT, top 90%)

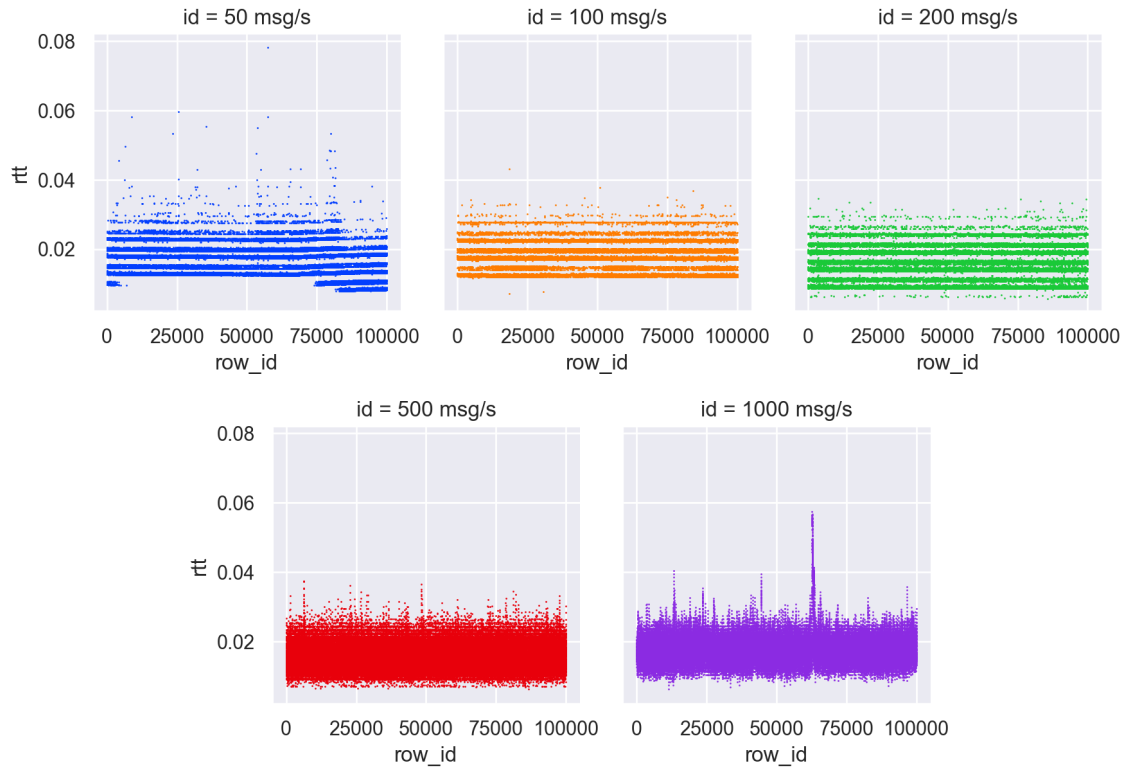


Figure A.1.4: 5G SA scatterplots

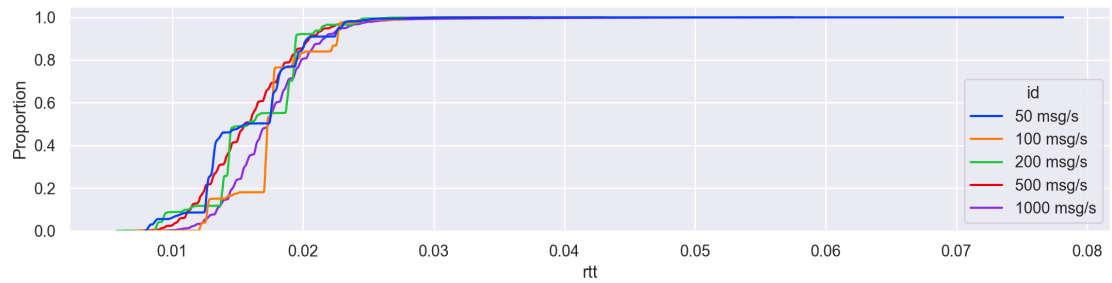


Figure A.1.5: 5G SA 6 ECDF

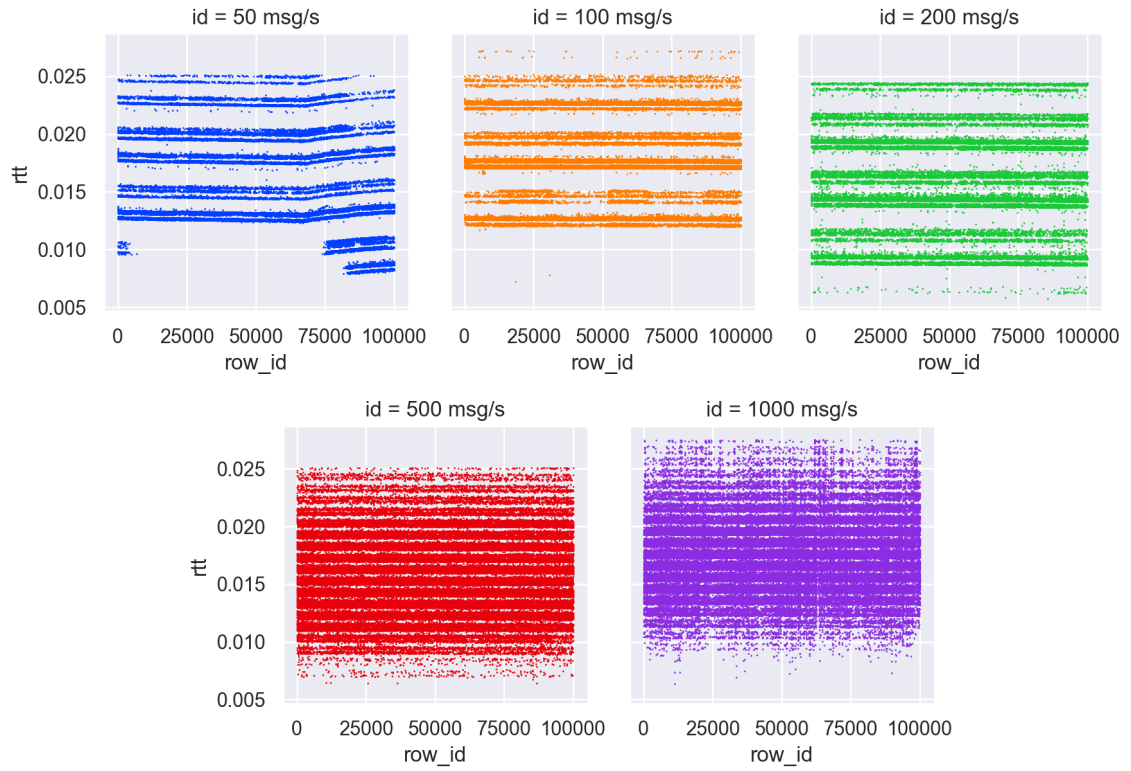


Figure A.1.6: 5G SA scatterplots (top 99%)

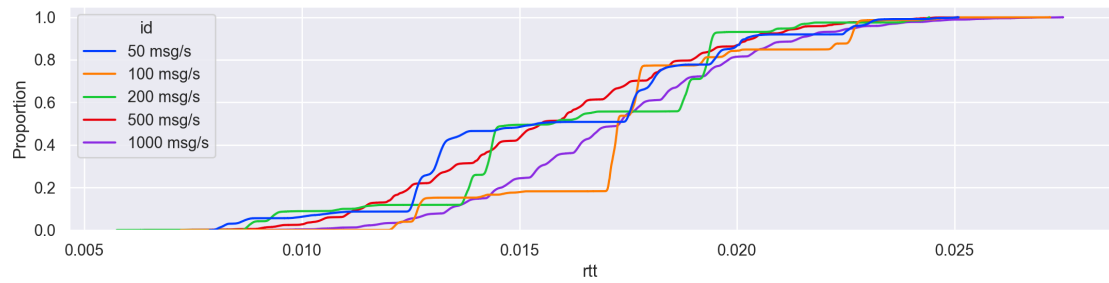


Figure A.1.7: 5G SA ECDF (top 99%)

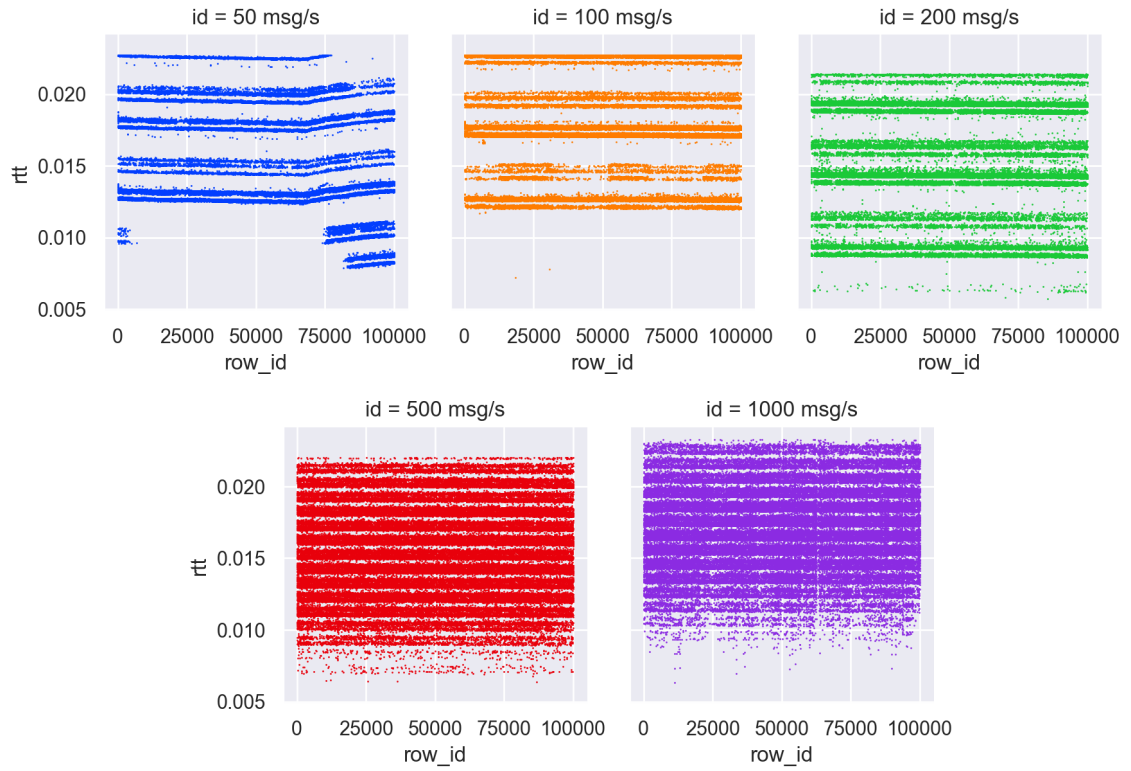


Figure A.1.8: 5G SA scatterplots (top 95%)

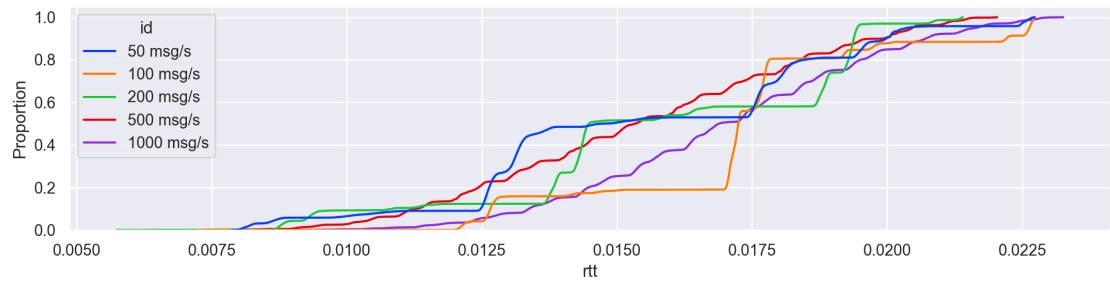


Figure A.1.9: 5G SA ECDF (top 95%)

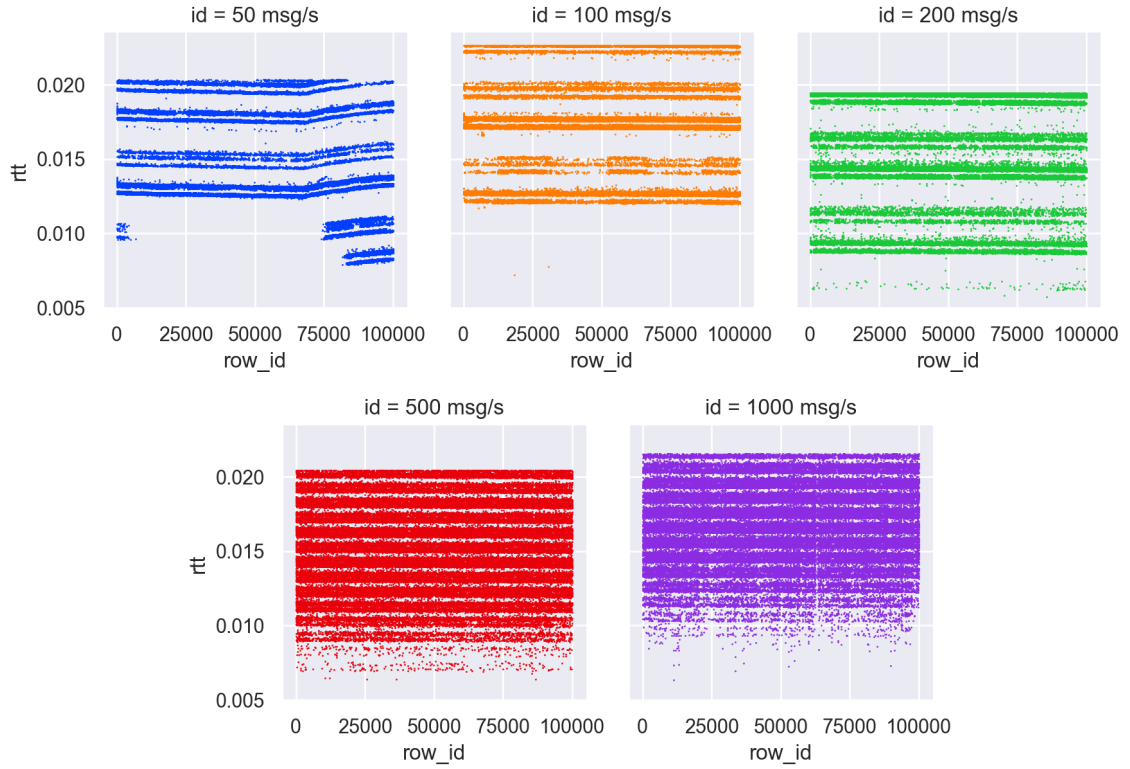


Figure A.1.10: 5G SA scatterplots (top 90%)

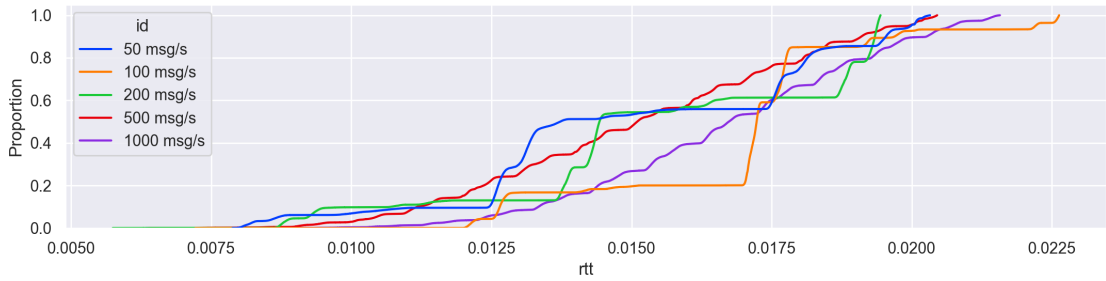


Figure A.1.11: 5G SA ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
50 msg/s	0.0159	0.0227	0.0155	0.0203	0.0151	0.0156
100 msg/s	0.0176	0.0227	0.0173	0.0226	0.0170	0.0173
200 msg/s	0.0163	0.0214	0.0159	0.0194	0.0156	0.0158
500 msg/s	0.0159	0.0220	0.0154	0.0205	0.0151	0.0155
1000 msg/s	0.0175	0.0233	0.0170	0.0216	0.0167	0.0173

Table A.1: RTT statistics (5G SA)

	max. pending	non-unique	mean jitter	max. jitter	timed out
50 msg/s	0.0856	24	0.0054	0.0656	0
100 msg/s	0.0331	11431	0.0033	0.0231	0
200 msg/s	0.0298	12185	0.0048	0.0248	0
500 msg/s	0.0259	9474	0.0030	0.0239	0
1000 msg/s	0.0230	10819	0.0016	0.0220	0

Table A.2: Delivery statistics (5G SA)

A.1.1 Different Payload Sizes

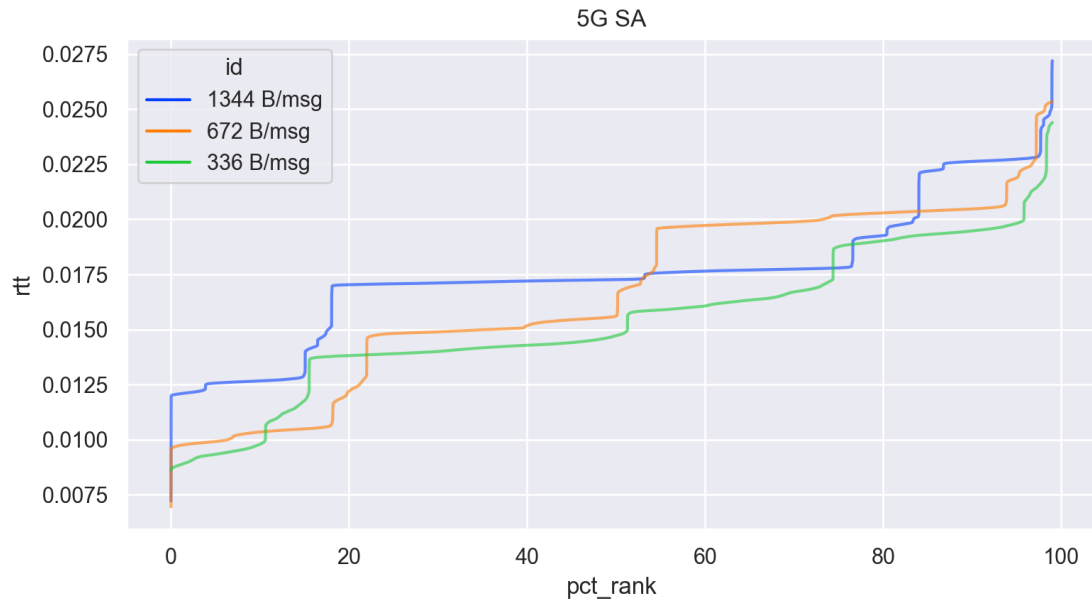


Figure A.1.12: 5G SA RTT ranked (top 99%)

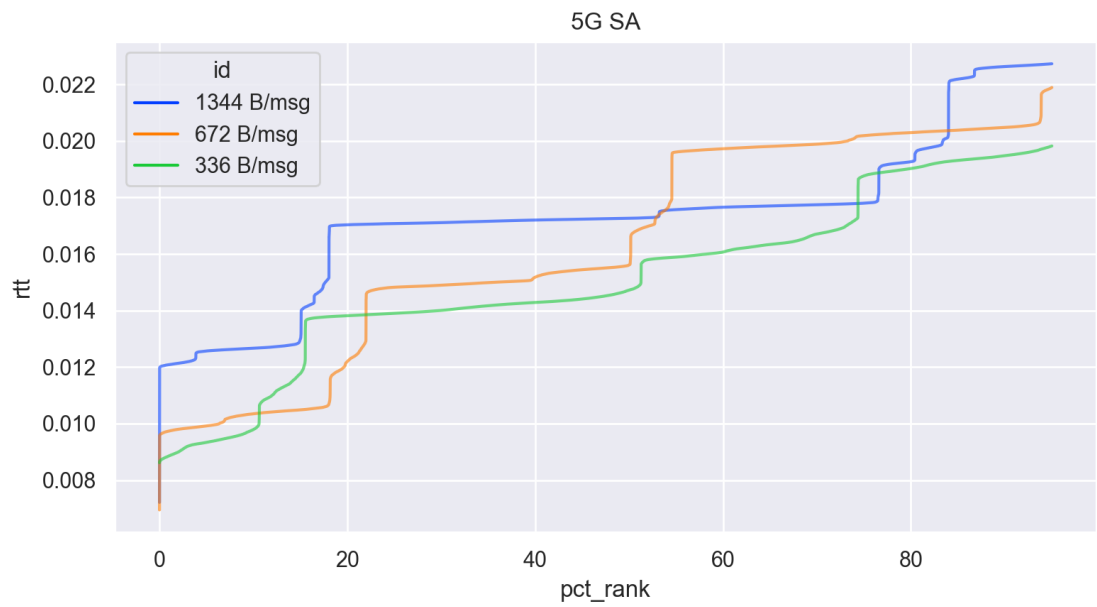


Figure A.1.13: 5G SA RTT ranked (top 95%)

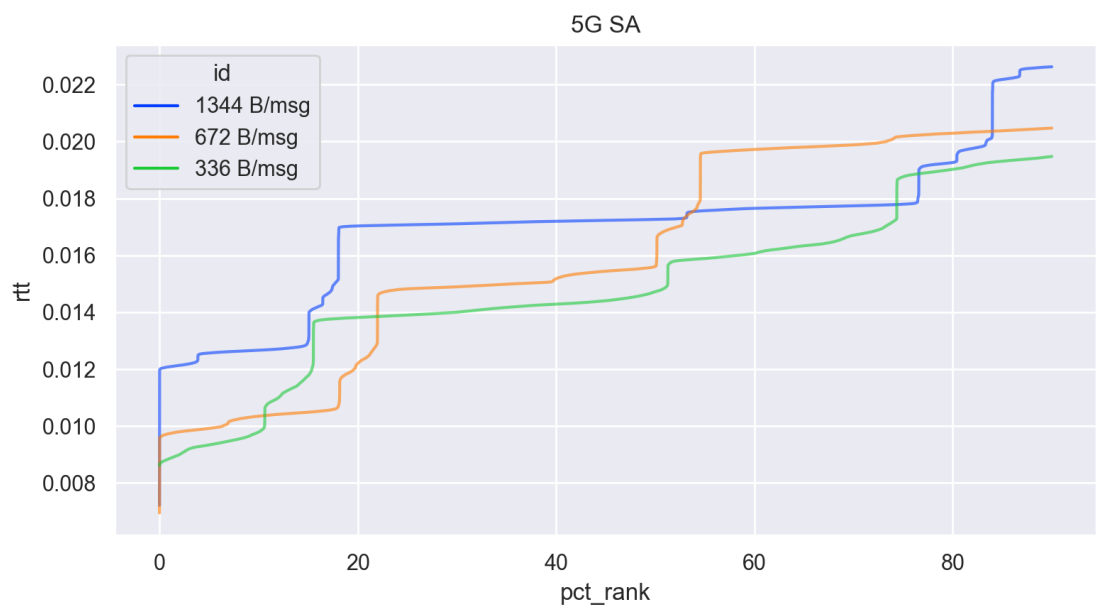


Figure A.1.14: 5G SA RTT ranked (top 90%)

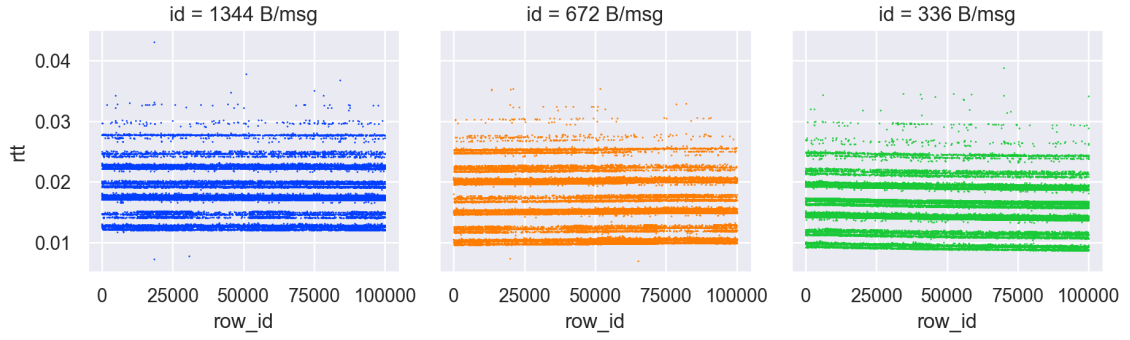


Figure A.1.15: 5G SA message size scatterplots

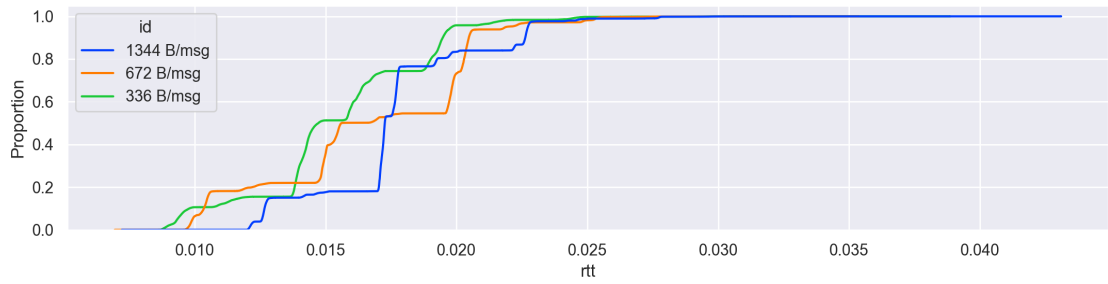


Figure A.1.16: 5G SA message size ECDF

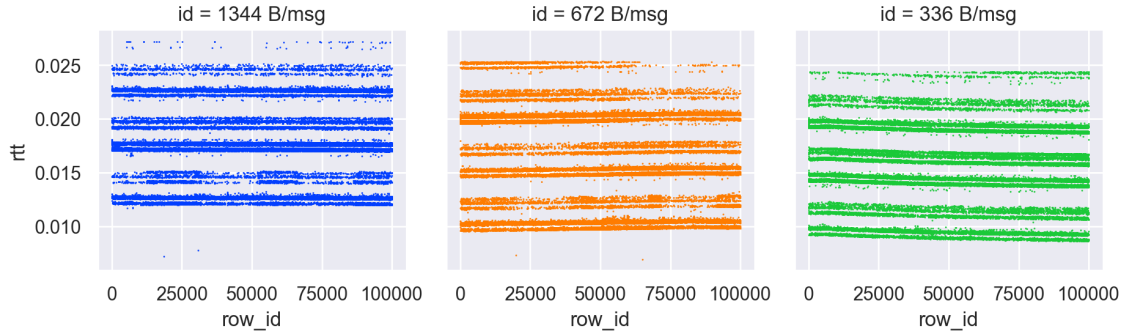


Figure A.1.17: 5G SA message size scatterplots (top 99%)

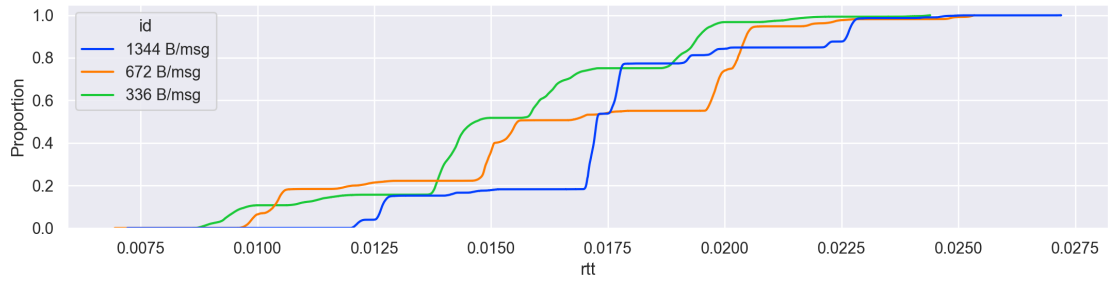


Figure A.1.18: 5G SA message size ECDF (top 99%)

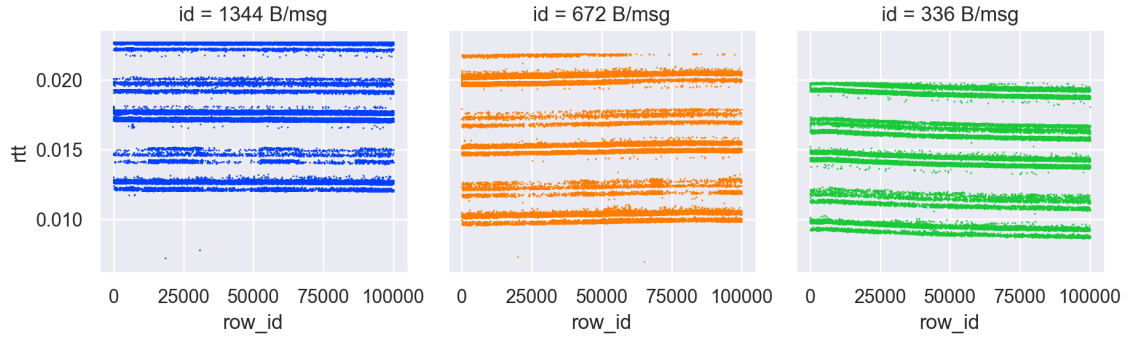


Figure A.1.19: 5G SA message size scatterplots (top 95%)

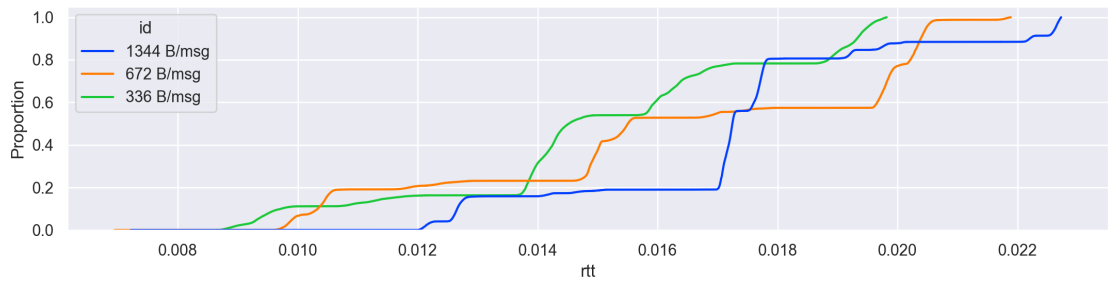


Figure A.1.20: 5G SA message size ECDF (top 95%)

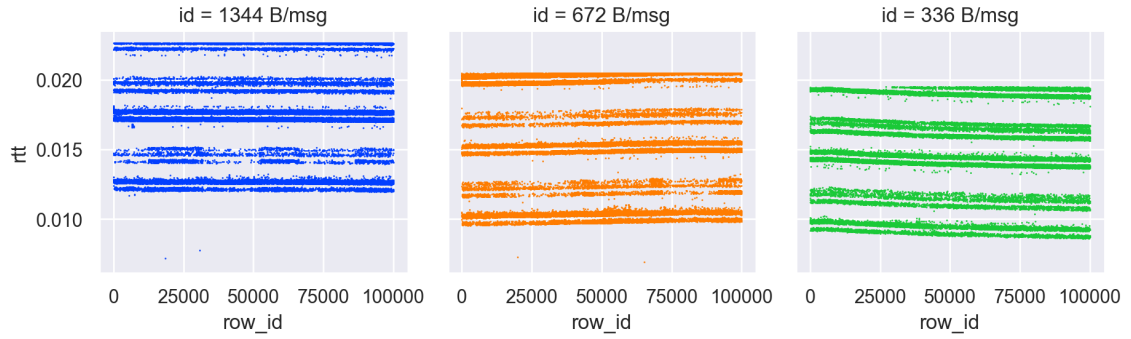


Figure A.1.21: 5G SA message size scatterplots (top 90%)

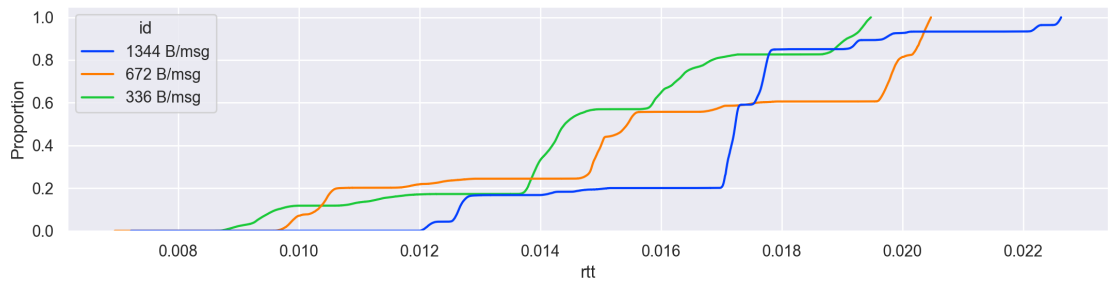


Figure A.1.22: 5G SA message size ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
1344 B/msg	0.0176	0.0227	0.0173	0.0226	0.0170	0.0173
672 B/msg	0.0167	0.0219	0.0163	0.0205	0.0160	0.0156
336 B/msg	0.0155	0.0198	0.0151	0.0195	0.0148	0.0147

Table A.3: RTT statistics (5G SA | Size)

	max. pending	non-unique	mean jitter	max. jitter	timed out
1344 B/msg	0.0331	11431	0.0033	0.0231	0
672 B/msg	0.0322	10472	0.0039	0.0222	0
336 B/msg	0.0375	8188	0.0040	0.0275	0

Table A.4: Delivery statistics (5G SA | Size)

A.2 5G NSA

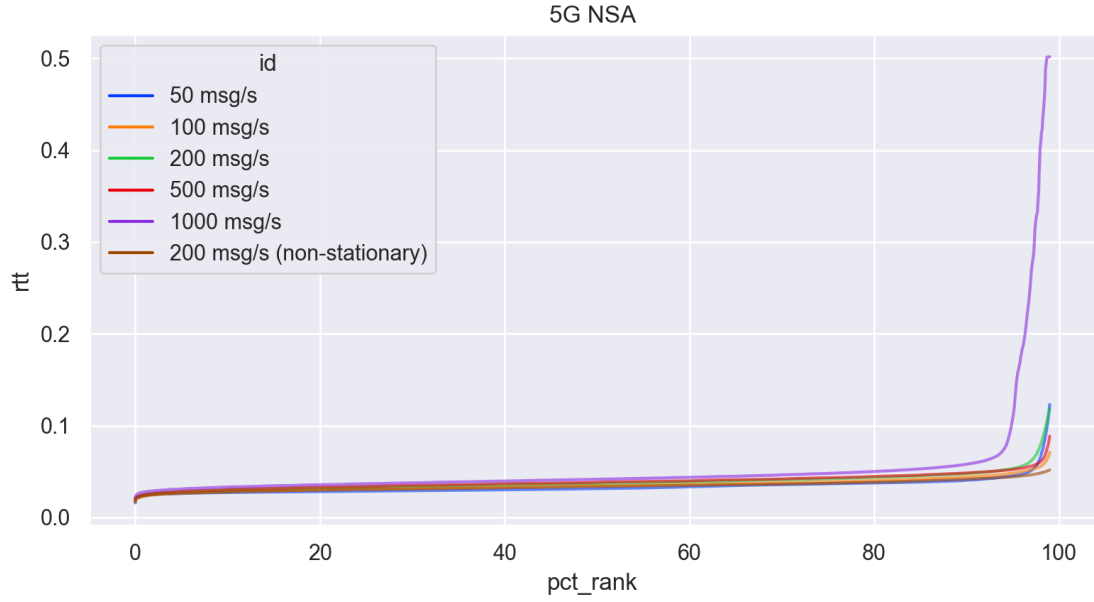


Figure A.2.1: 5G NSA (Rank to RTT, top 95%)

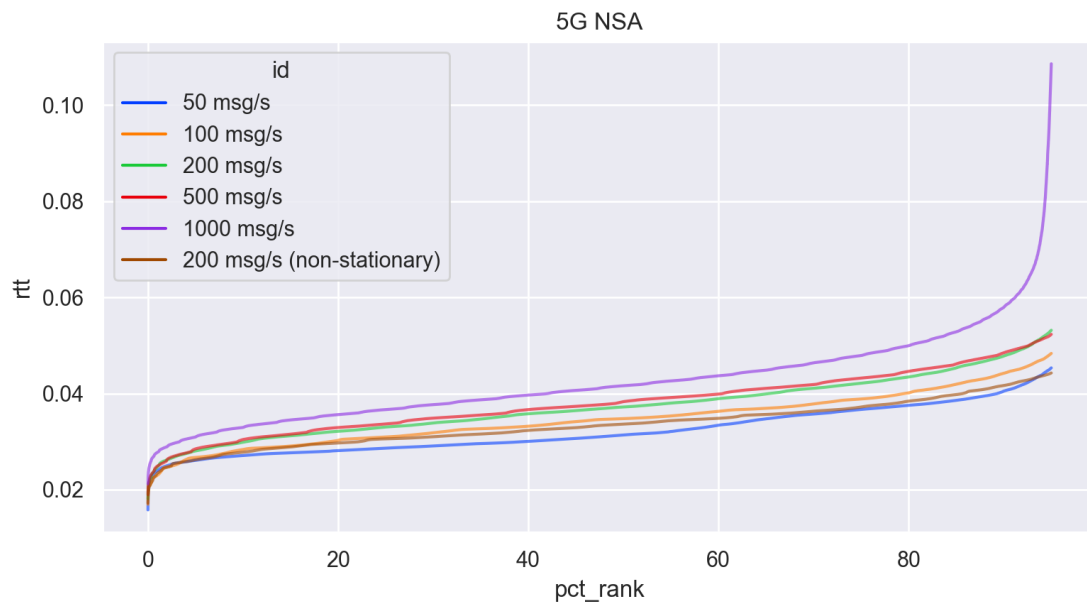


Figure A.2.2: 5 NSA (Rank to RTT, top 95%)

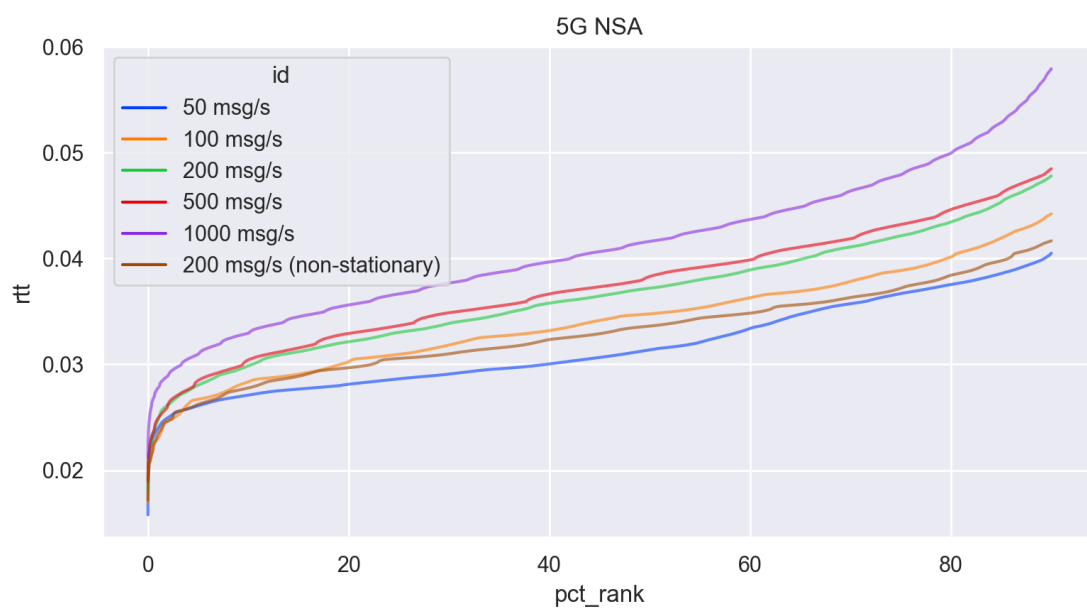


Figure A.2.3: 5 NSA (Rank to RTT, top 90%)

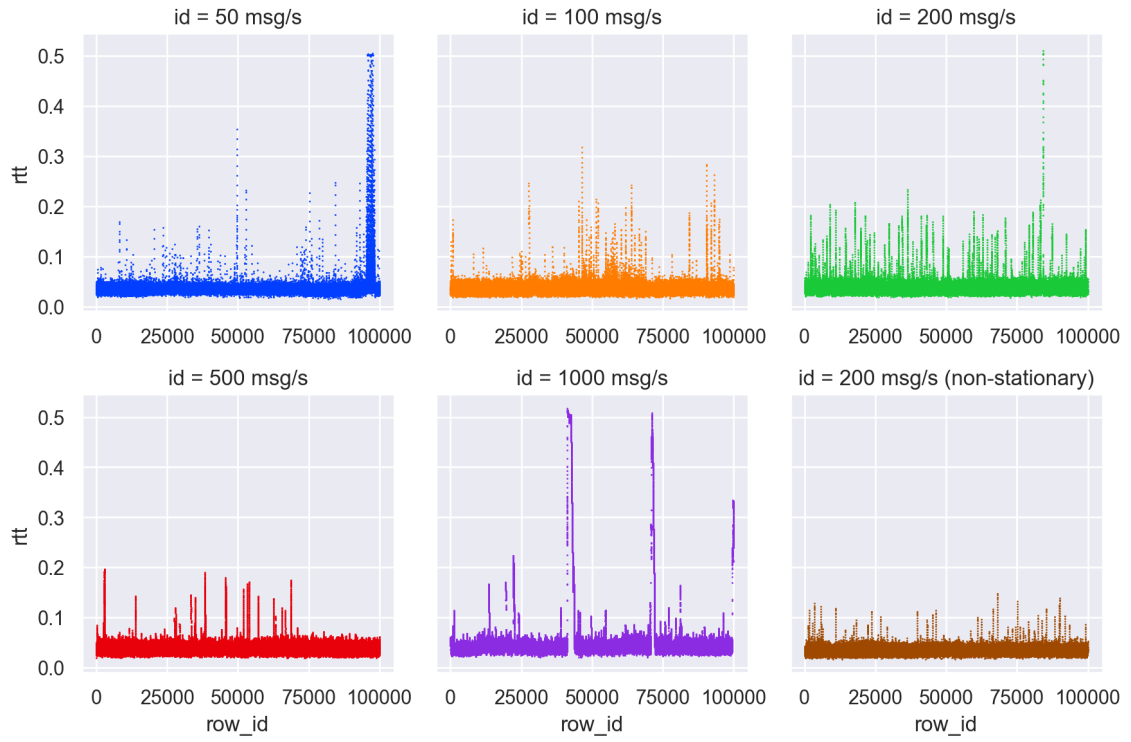


Figure A.2.4: 5G NSA scatterplots

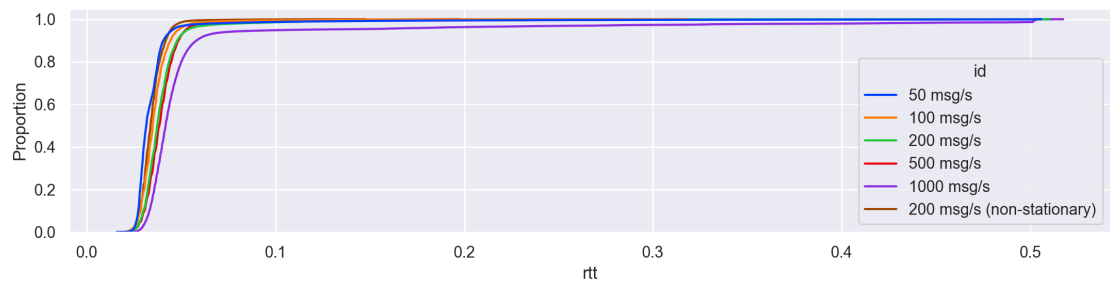


Figure A.2.5: 5G NSA ECDF

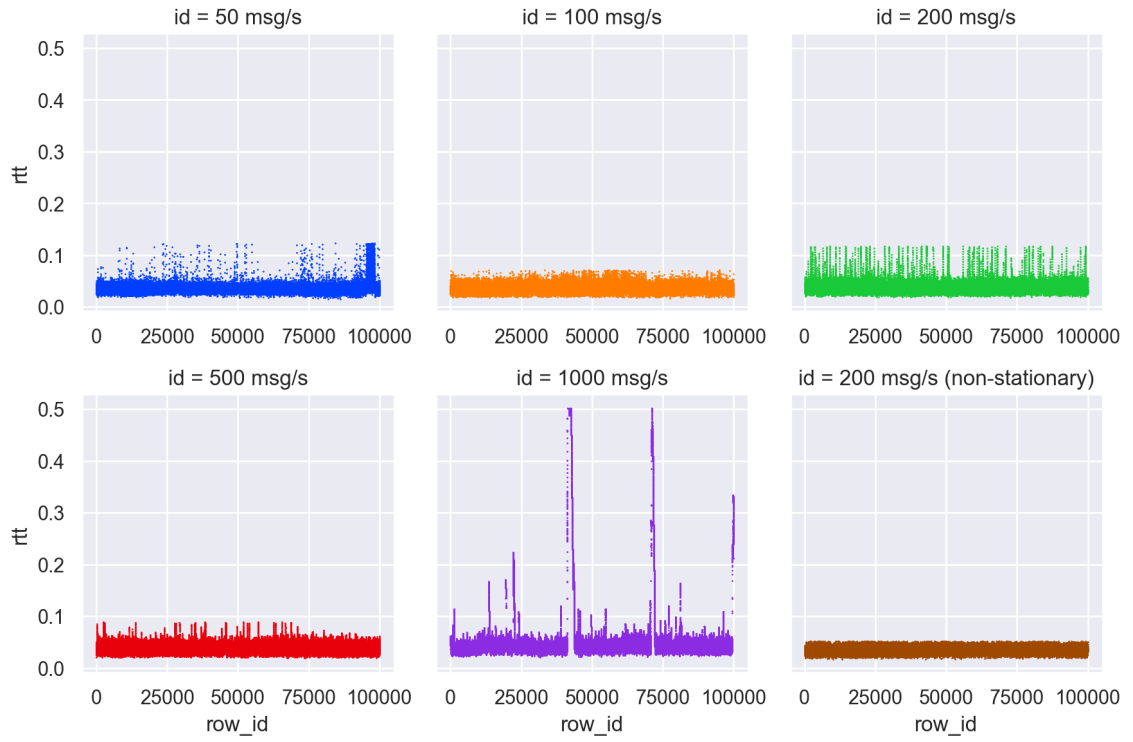


Figure A.2.6: 5G NSA scatterplots (top 99%)

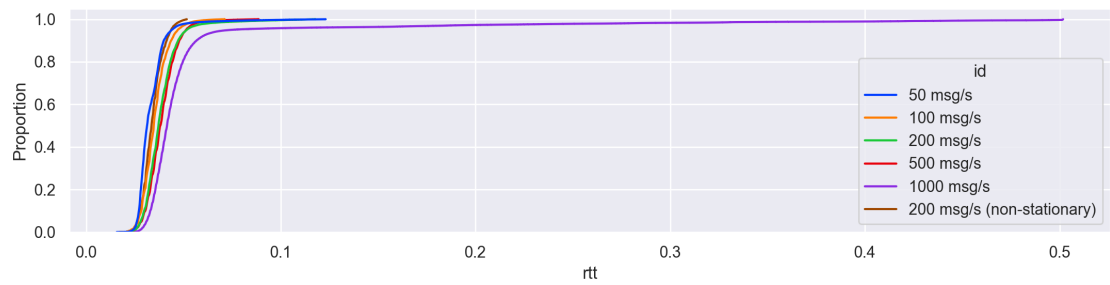


Figure A.2.7: 5G NSA ECDF (top 99%)

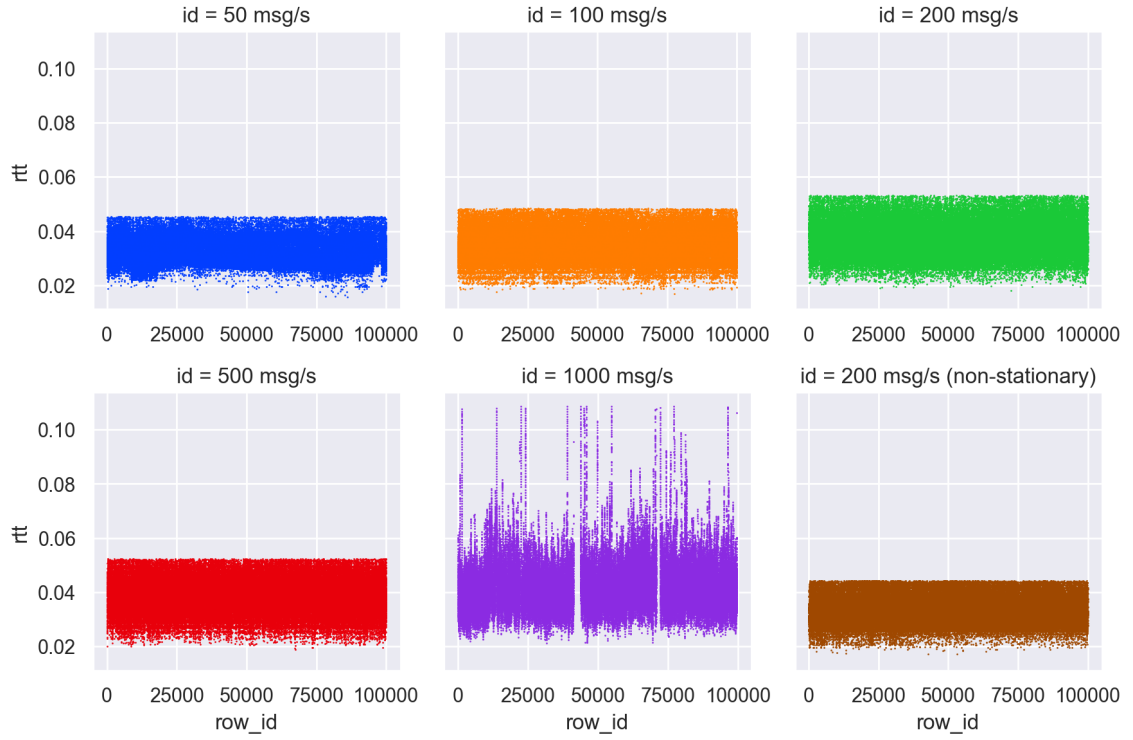


Figure A.2.8: 5G NSA scatterplots (top 95%)

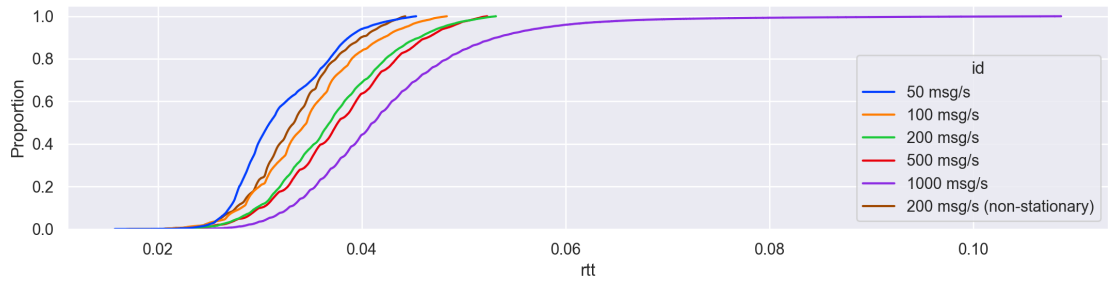


Figure A.2.9: 5G NSA ECDF (top 95%)

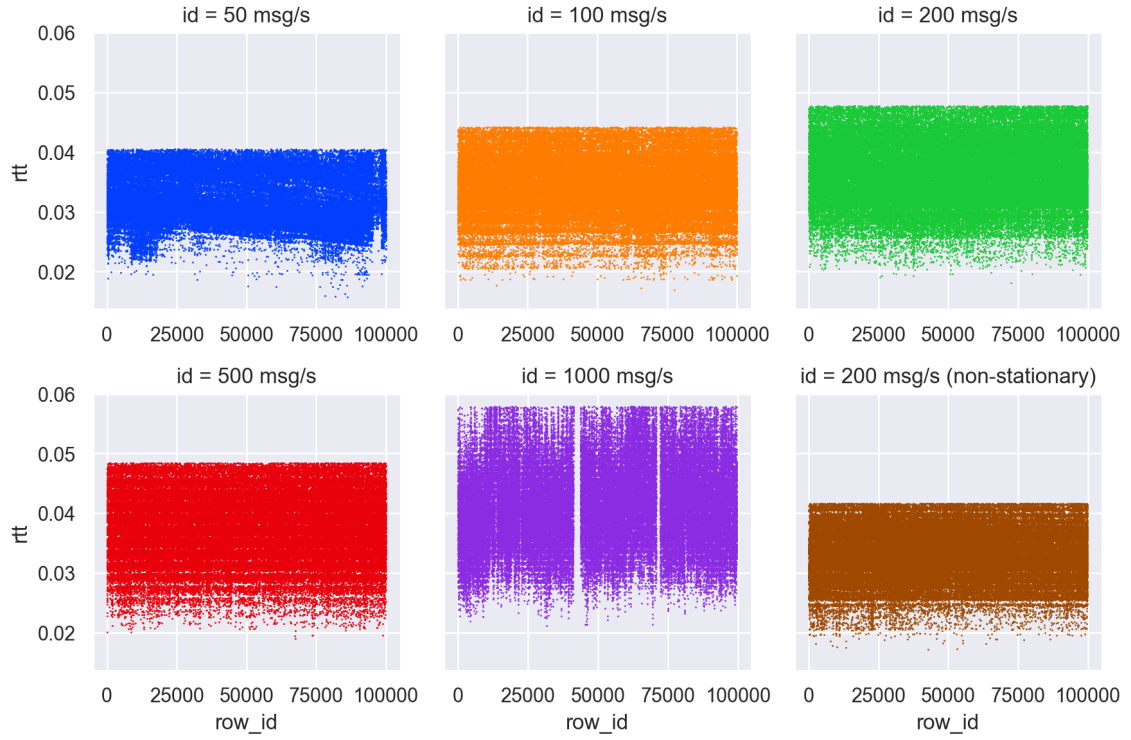


Figure A.2.10: 5G NSA scatterplots (top 90%)

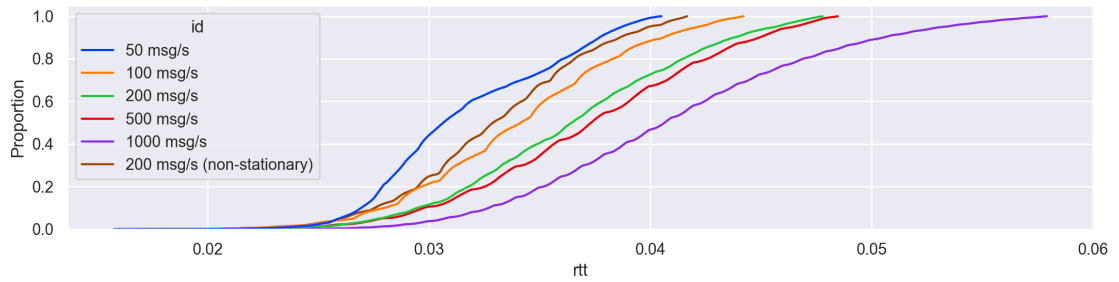


Figure A.2.11: 5G NSA ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
50 msg/s	0.0356	0.0453	0.0322	0.0405	0.0316	0.0314
100 msg/s	0.0363	0.0483	0.0346	0.0442	0.0340	0.0348
200 msg/s	0.0397	0.0532	0.0372	0.0478	0.0365	0.0372
500 msg/s	0.0397	0.0523	0.0380	0.0485	0.0373	0.0383
1000 msg/s	0.0570	0.1087	0.0425	0.0579	0.0411	0.0416
200 msg/s ^{moving}	0.0344	0.0443	0.0335	0.0417	0.0330	0.0336

Table A.5: RTT statistics (5G NSA)

	max. pending	non-unique	mean jitter	max. jitter	timed out
50 msg/s	0.4914	2222	0.0069	0.4714	98
100 msg/s	0.2692	7854	0.0056	0.2592	0
200 msg/s	0.1497	15835	0.0047	0.1414	4
500 msg/s	0.1313	14515	0.0030	0.1293	0
1000 msg/s	0.1281	16146	0.0017	0.1241	1343
200 msg/s ^{moving}	0.1058	14581	0.0039	0.0987	0

Table A.6: Delivery statistics (5G NSA)

A.2.1 Different Payload Sizes

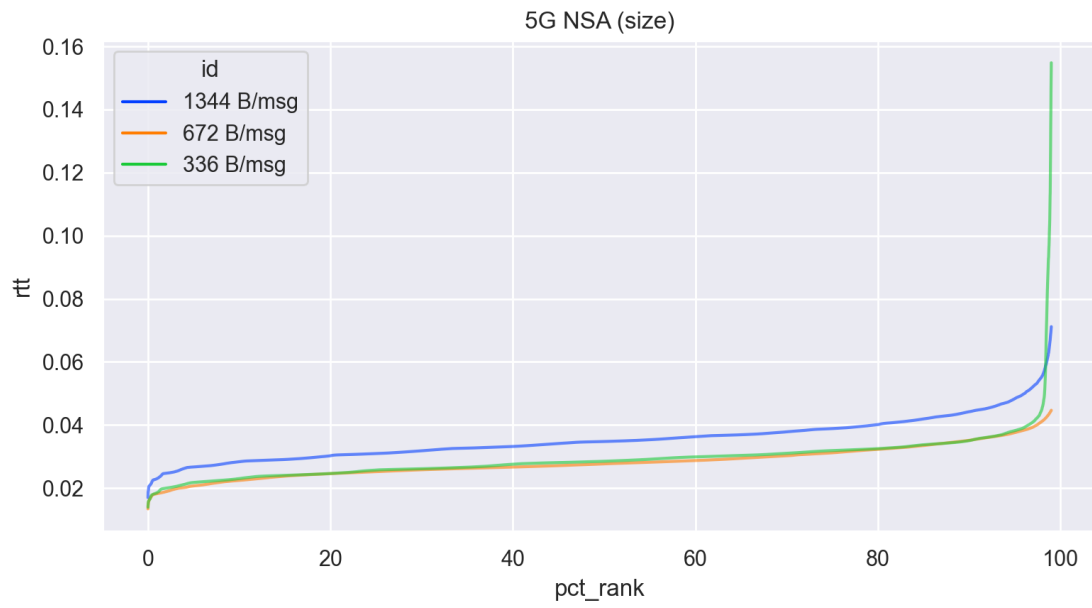


Figure A.2.12: 5G NSA RTT ranked (top 99%)

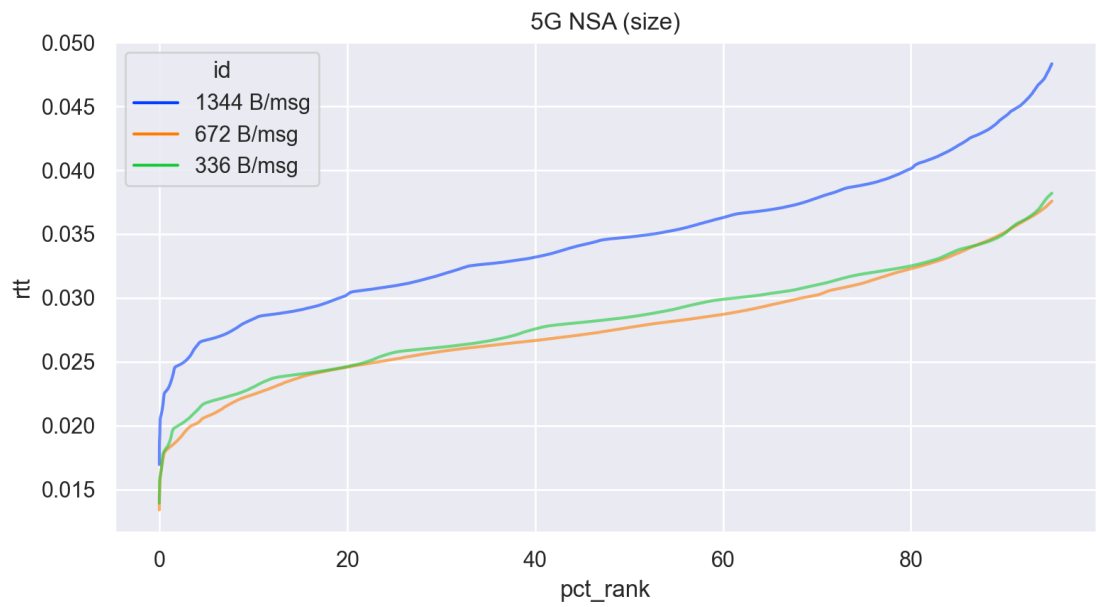


Figure A.2.13: 5G NSA RTT ranked (top 95%)

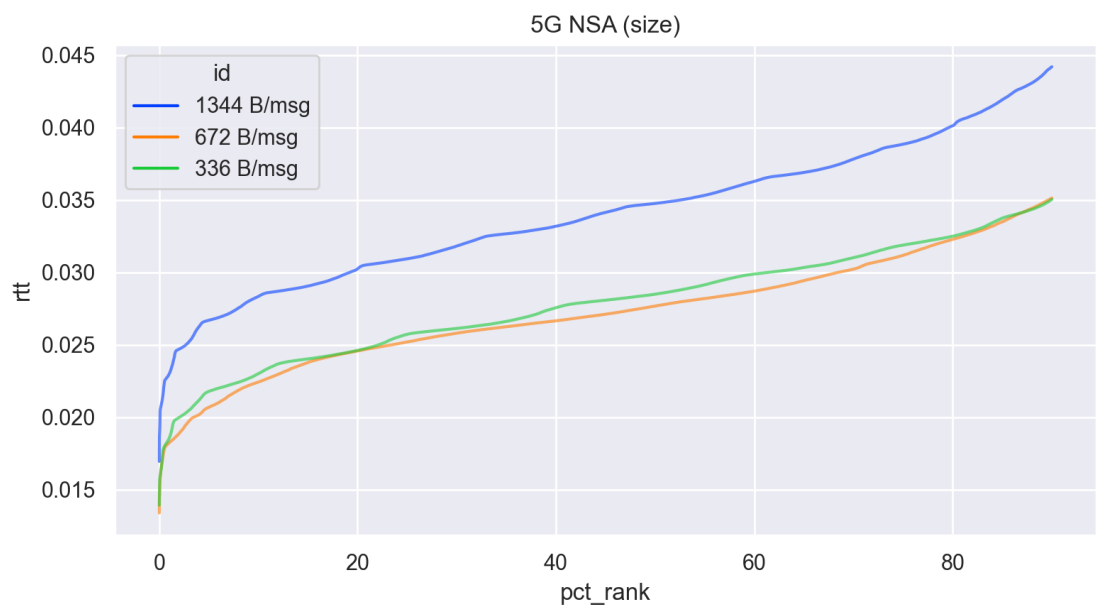


Figure A.2.14: 5G NSA RTT ranked (top 90%)

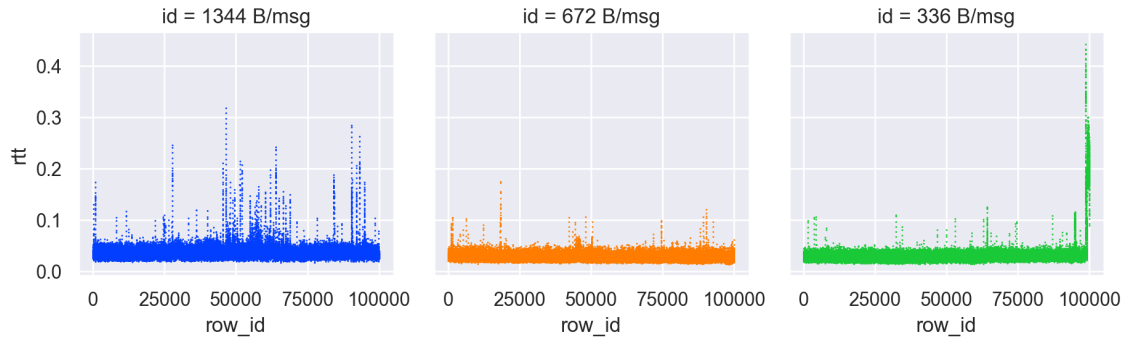


Figure A.2.15: 5G NSA message size scatterplots

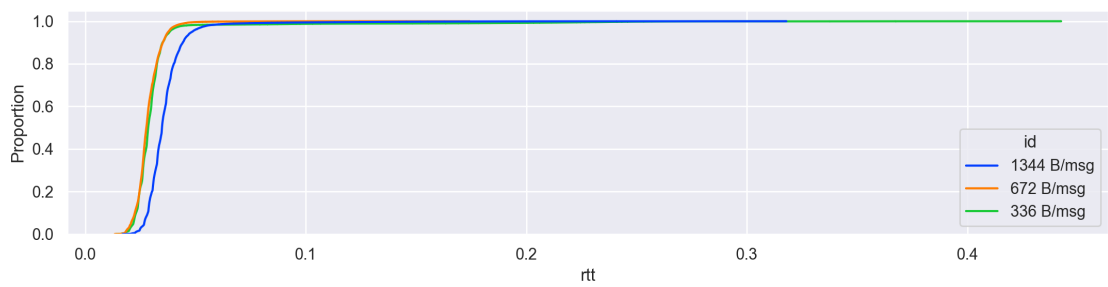


Figure A.2.16: 5G NSA message size ECDF

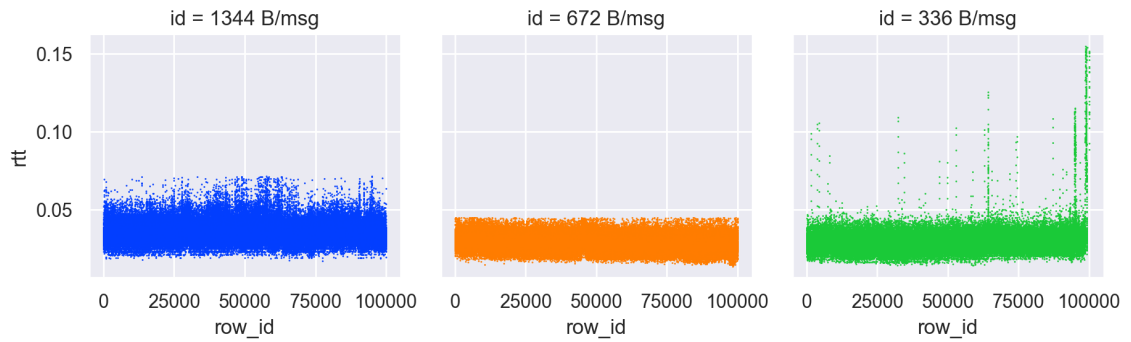


Figure A.2.17: 5G NSA message size scatterplots (top 99%)

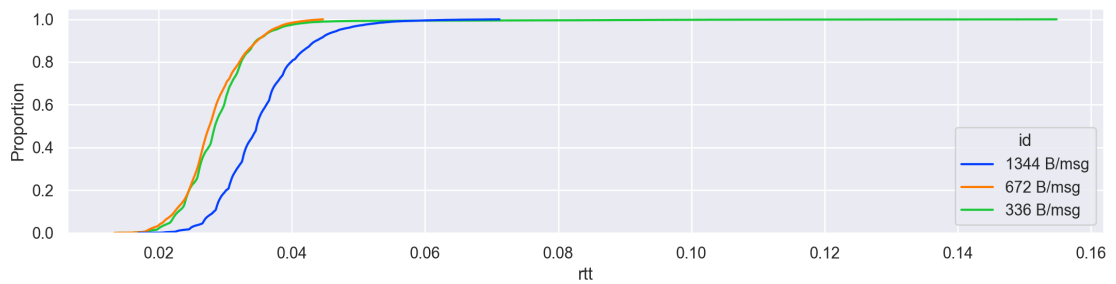


Figure A.2.18: 5G NSA message size ECDF (top 99%)

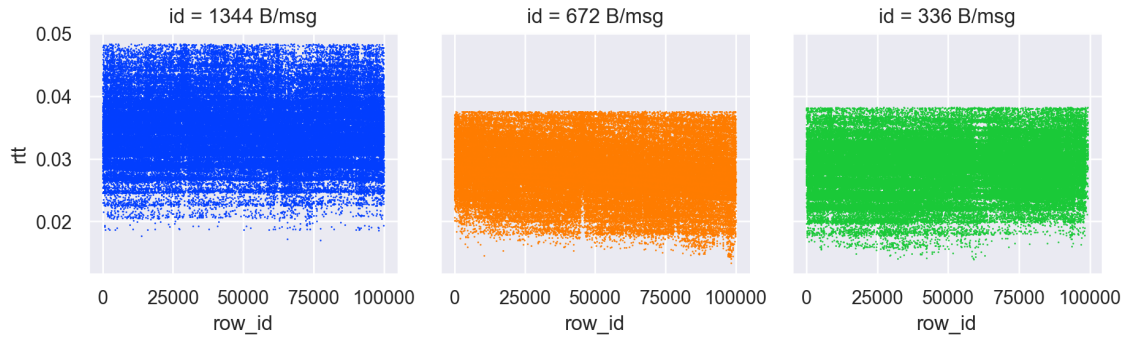


Figure A.2.19: 5G NSA message size scatterplots (top 95%)

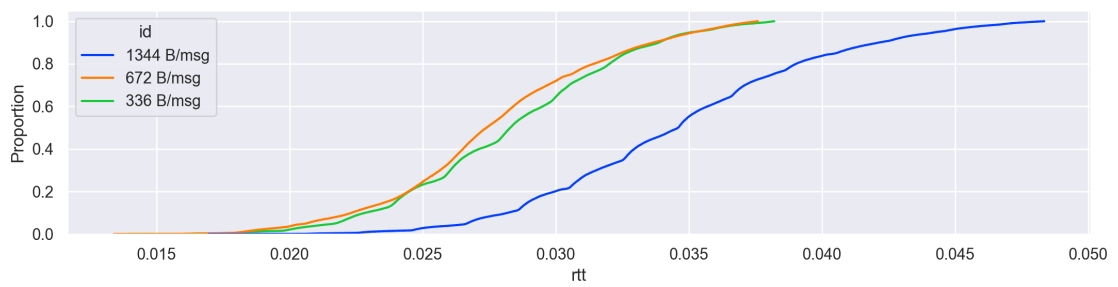


Figure A.2.20: 5G NSA message size ECDF (top 95%)

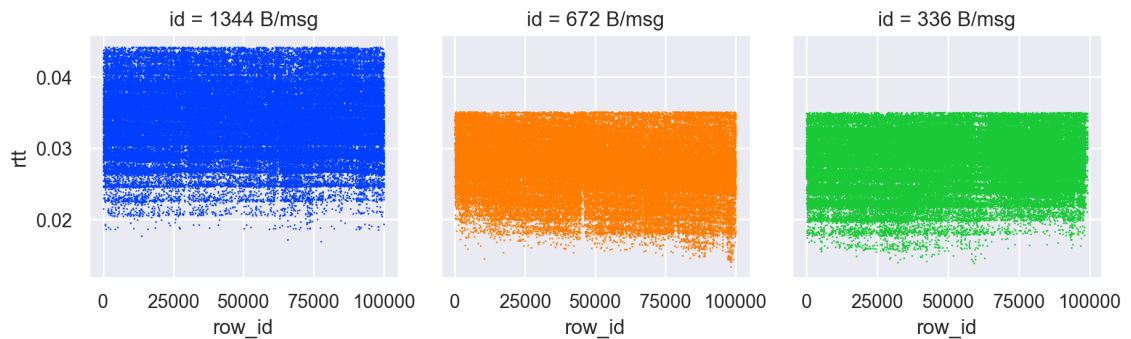


Figure A.2.21: 5G NSA message size scatterplots (top 90%)

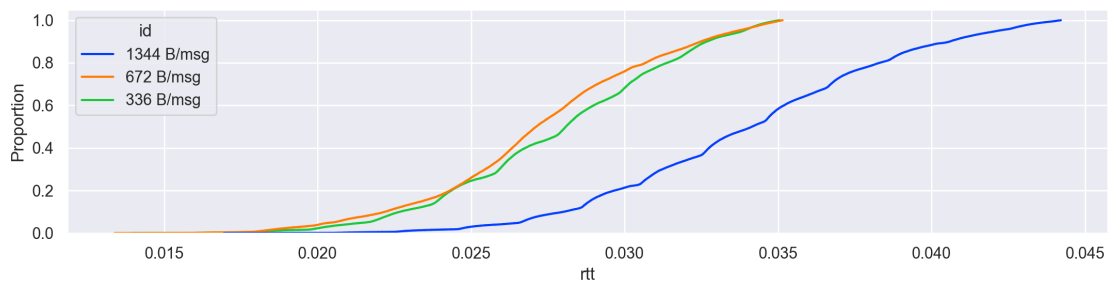


Figure A.2.22: 5G NSA message size ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
1344 B/msg	0.0363	0.0483	0.0346	0.0442	0.0340	0.0348
672 B/msg	0.0285	0.0376	0.0277	0.0351	0.0272	0.0277
336 B/msg	0.0312	0.0382	0.0283	0.0350	0.0278	0.0285

Table A.7: RTT statistics (5G NSA | Size)

	max. pending	non-unique	mean jitter	max. jitter	timed out
1344 B/msg	0.2692	7854	0.0056	0.2592	0
672 B/msg	0.0939	2602	0.0047	0.0846	0
336 B/msg	0.1456	4672	0.0045	0.1356	0

Table A.8: Delivery statistics (5G NSA | Size)

A.3 Wi-Fi 5

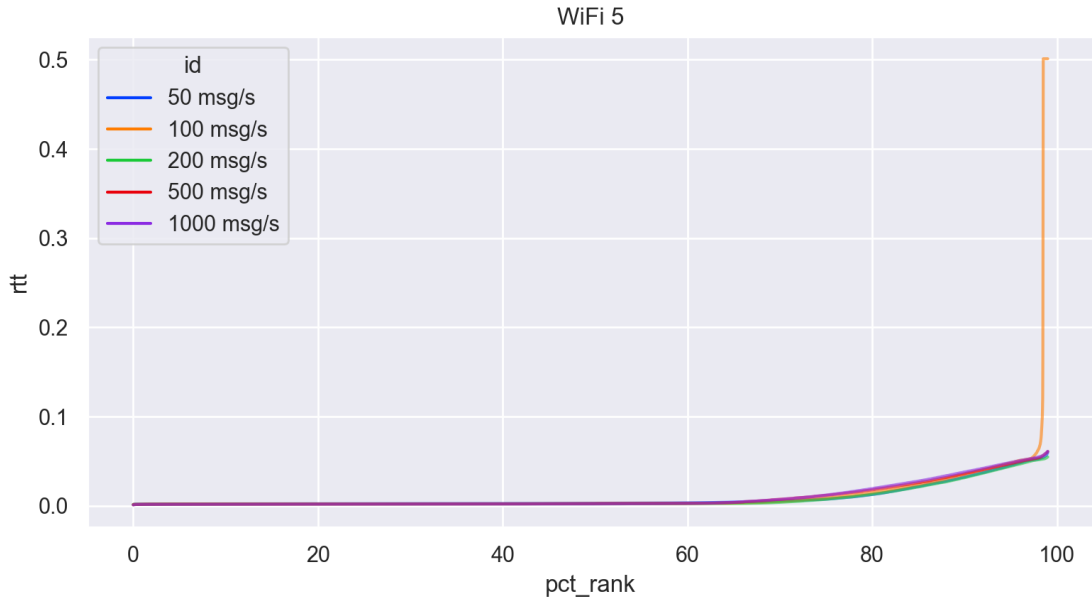


Figure A.3.1: Wi-Fi 5 (Rank to RTT, top 99%)

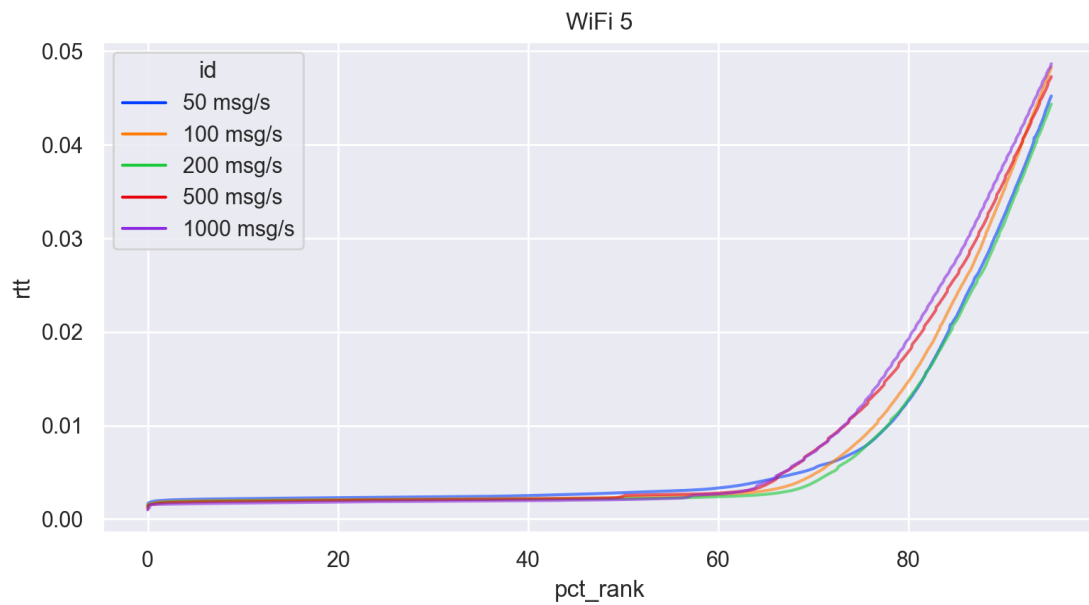


Figure A.3.2: Wi-Fi 5 (Rank to RTT, top 95%)

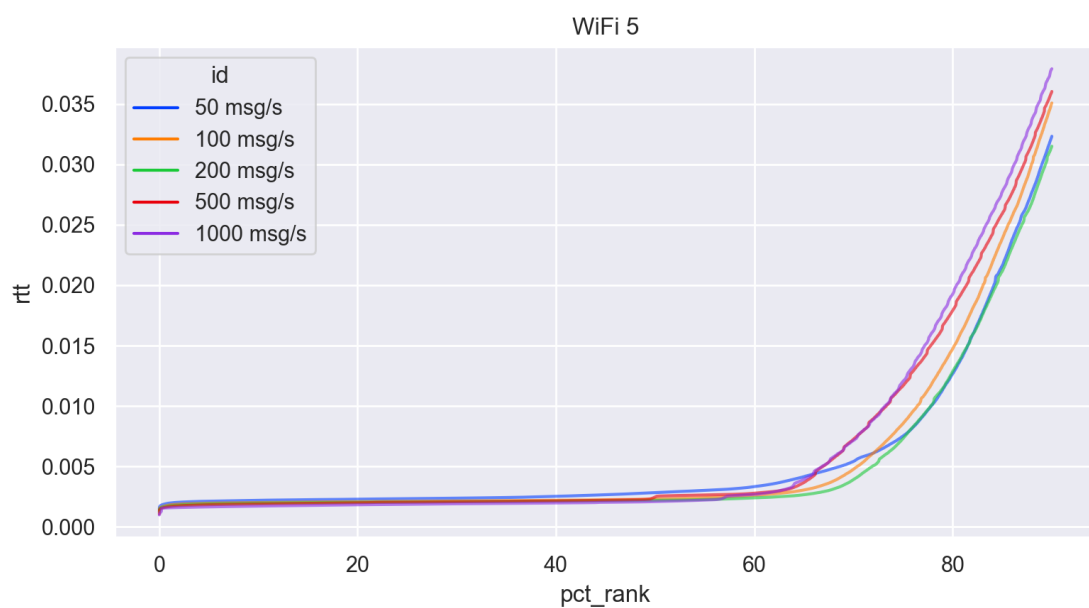


Figure A.3.3: Wi-Fi 5 (Rank to RTT, top 90%)

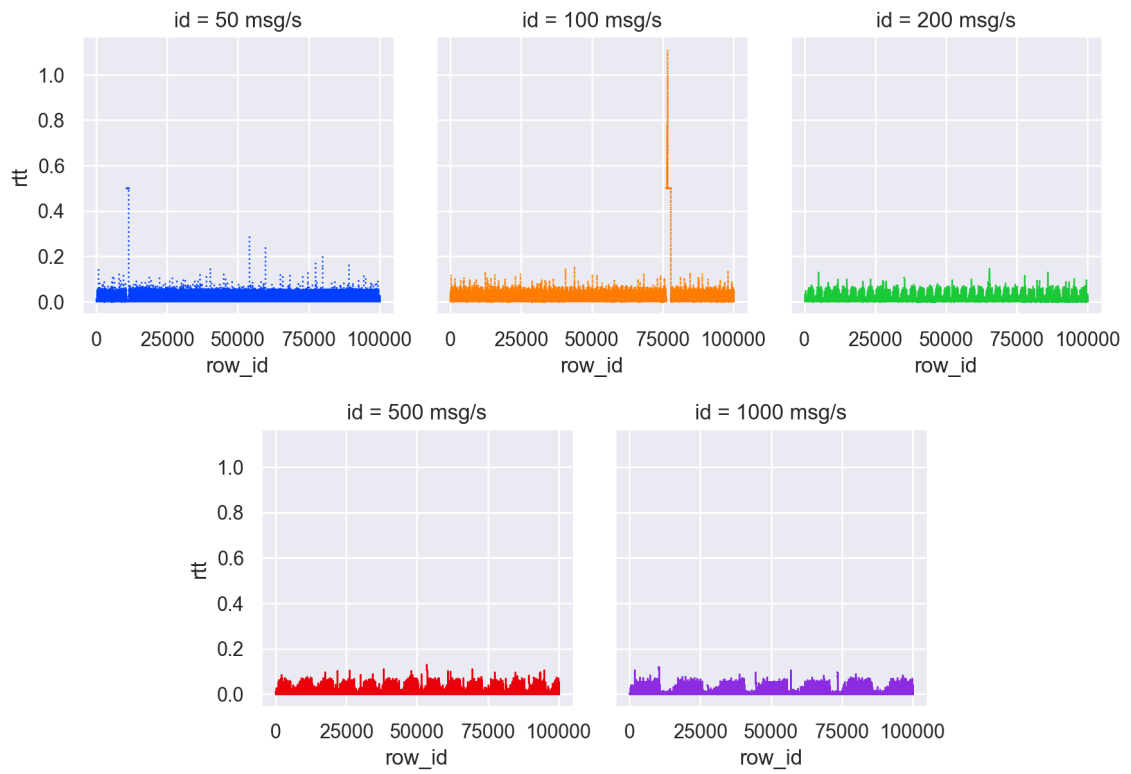


Figure A.3.4: Wi-Fi 5 scatterplots

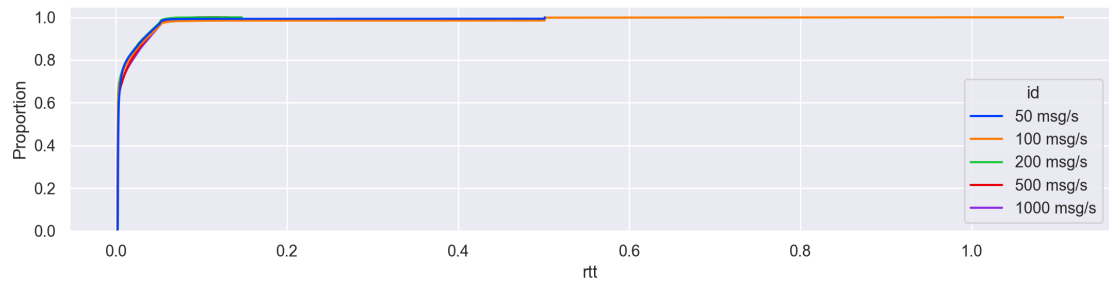


Figure A.3.5: Wi-Fi 5 ECDF

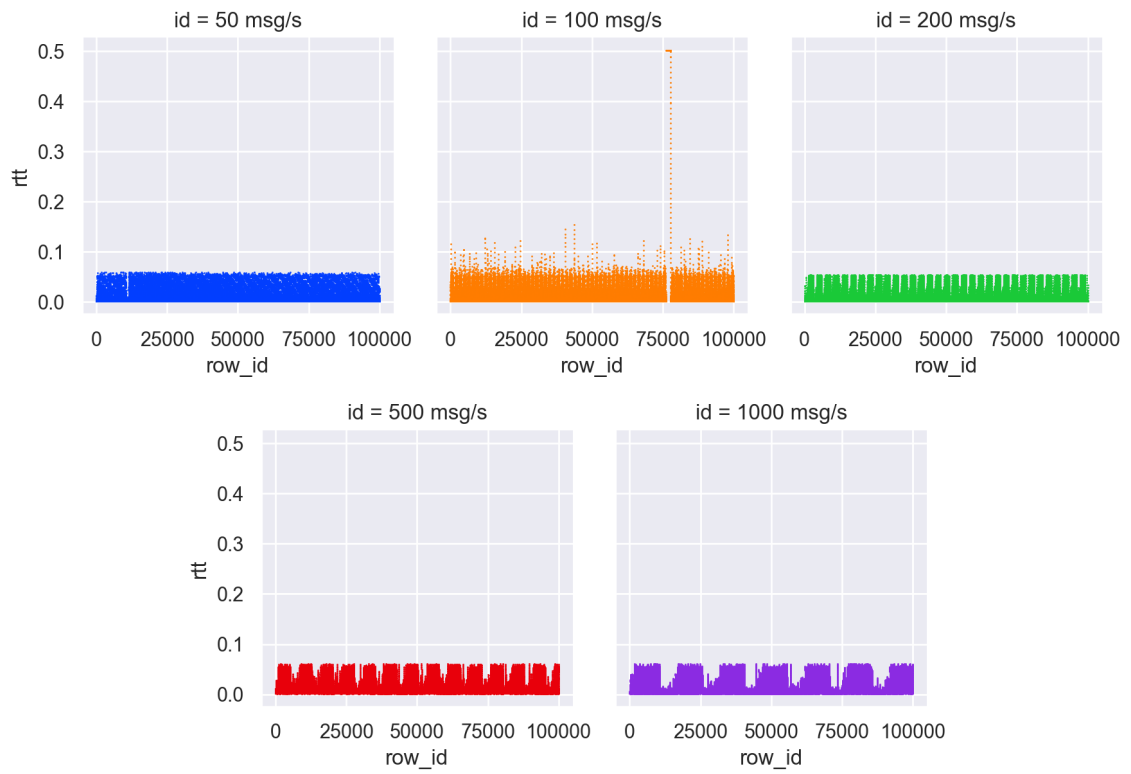


Figure A.3.6: Wi-Fi 5 scatterplots (top 99%)

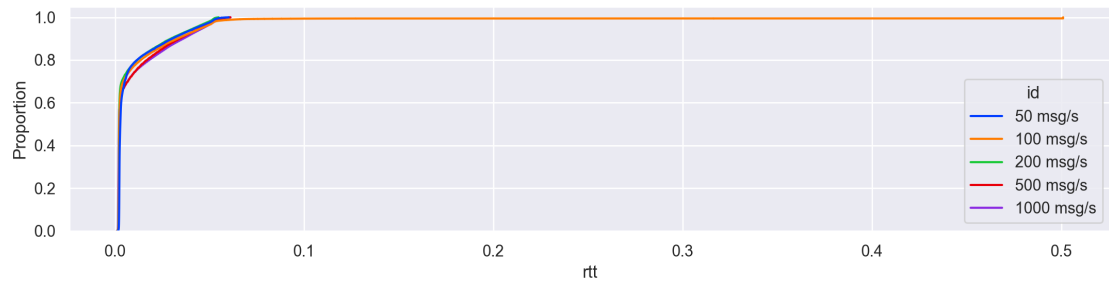


Figure A.3.7: Wi-Fi 5 ECDF (top 99%)

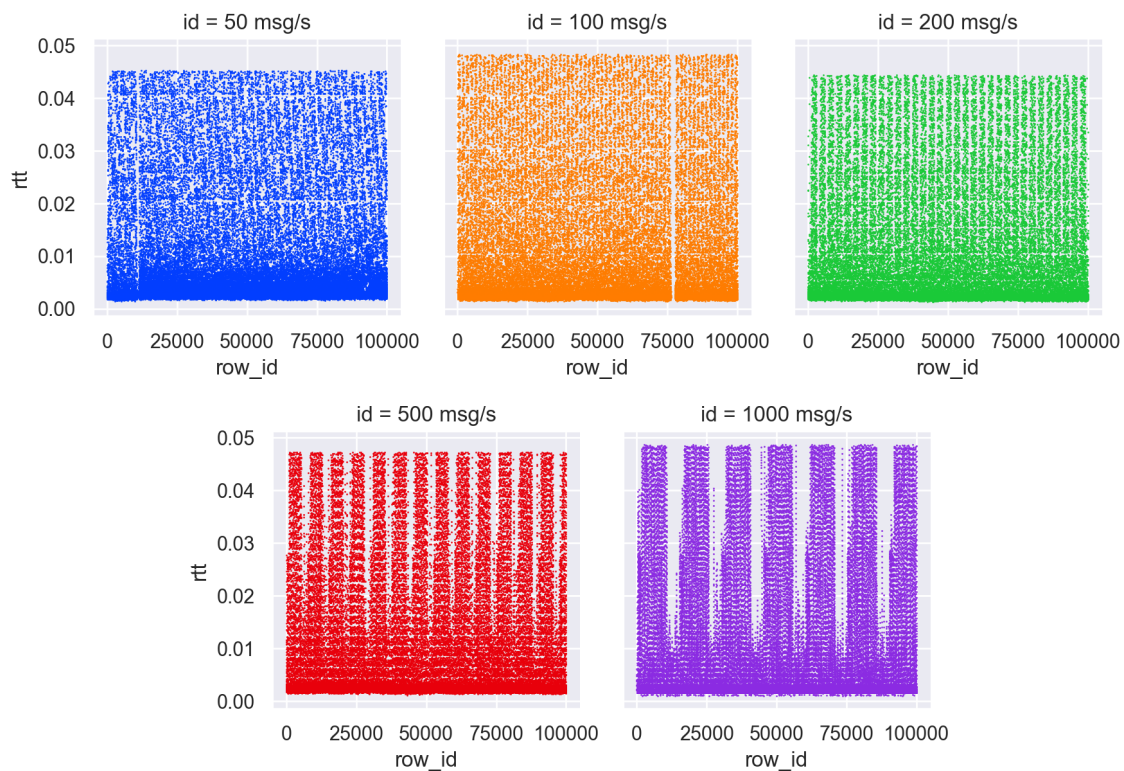


Figure A.3.8: Wi-Fi 5 scatterplots (top 95%)

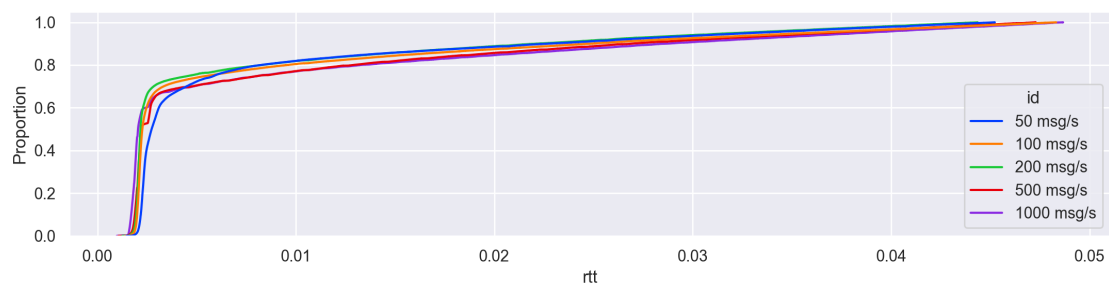


Figure A.3.9: Wi-Fi 5 ECDF (top 95%)

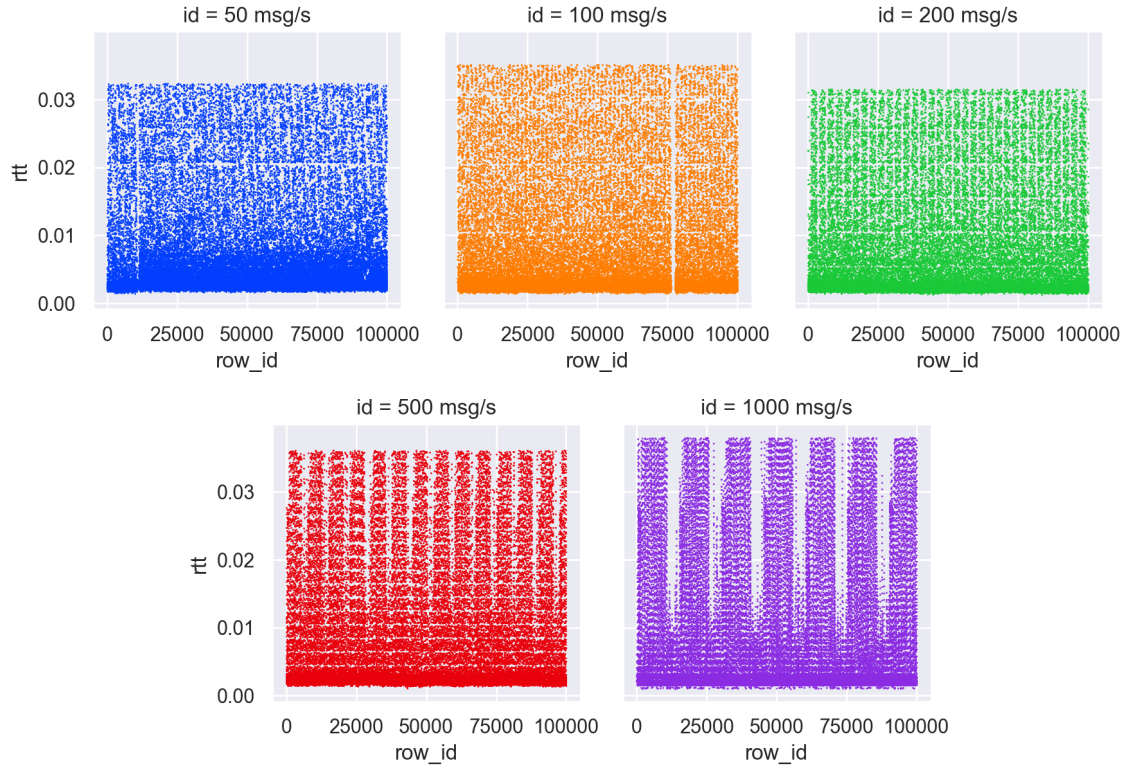


Figure A.3.10: Wi-Fi 5 scatterplots (top 90%)

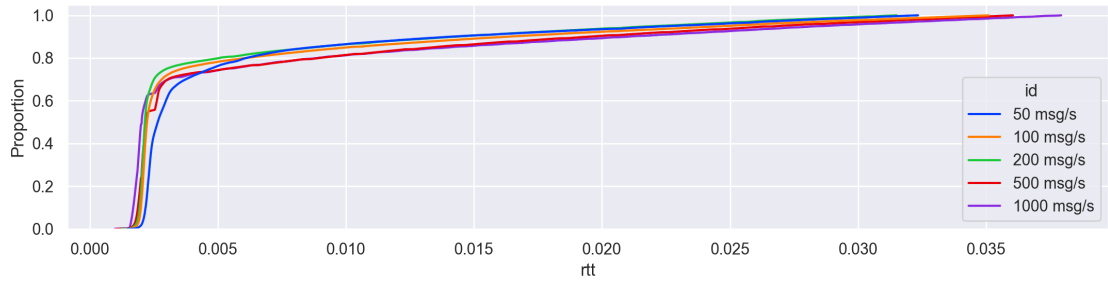


Figure A.3.11: Wi-Fi 5 ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
50 msg/s	0.0124	0.0452	0.0072	0.0323	0.0054	0.0028
100 msg/s	0.0169	0.0483	0.0074	0.0351	0.0055	0.0023
200 msg/s	0.0090	0.0444	0.0067	0.0315	0.0049	0.0022
500 msg/s	0.0104	0.0473	0.0080	0.0360	0.0061	0.0025
1000 msg/s	0.0106	0.0487	0.0082	0.0379	0.0063	0.0021

Table A.9: Wi-Fi 5 RTT statistics

	max. pending	non-unique	mean jitter	max. jitter	timed out
50 msg/s	0.5189	4512	0.0086	0.4989	649
100 msg/s	0.5809	4612	0.0053	0.5609	1493
200 msg/s	0.1336	4383	0.0030	0.1286	0
500 msg/s	0.1308	6416	0.0016	0.1288	0
1000 msg/s	0.1072	7916	0.0009	0.1062	0

Table A.10: Wi-Fi 5 delivery statistics

A.3.1 Different Payload Sizes

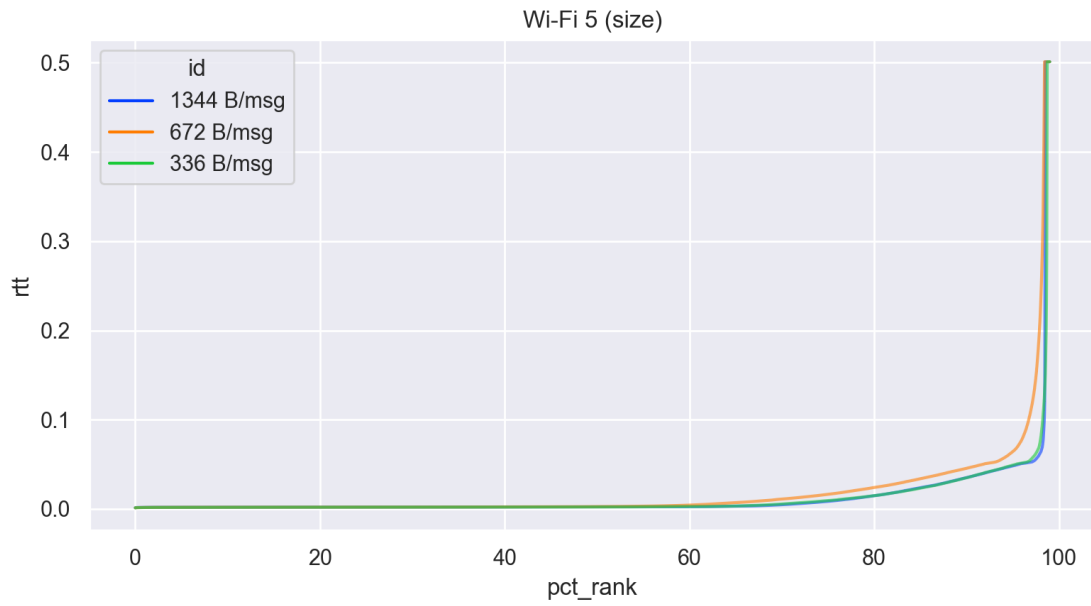


Figure A.3.12: Wi-Fi 5 RTT ranked (top 99%)

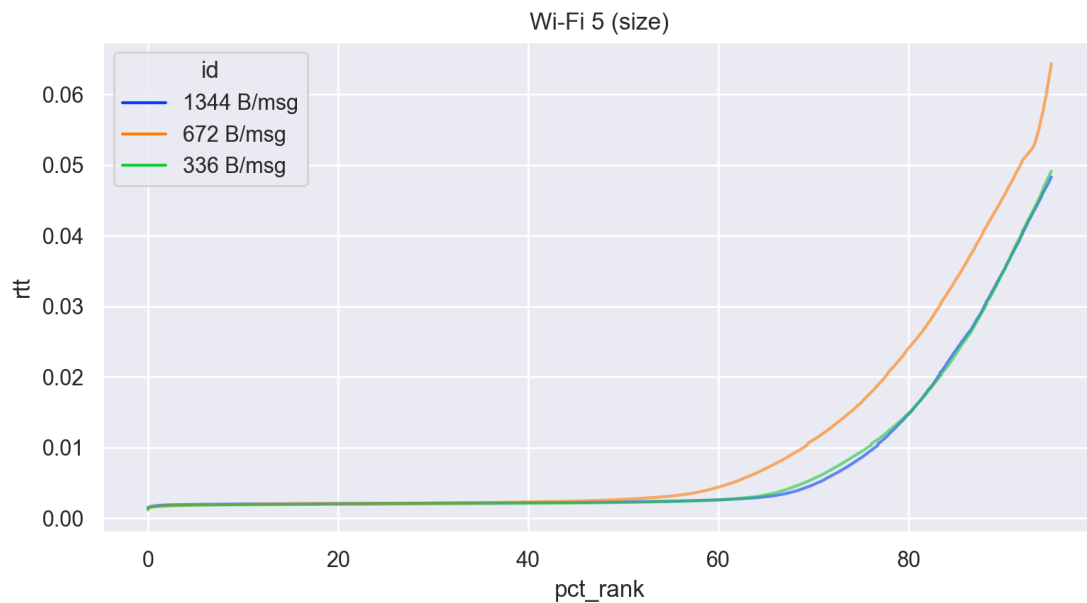


Figure A.3.13: Wi-Fi 5 RTT ranked (top 95%)

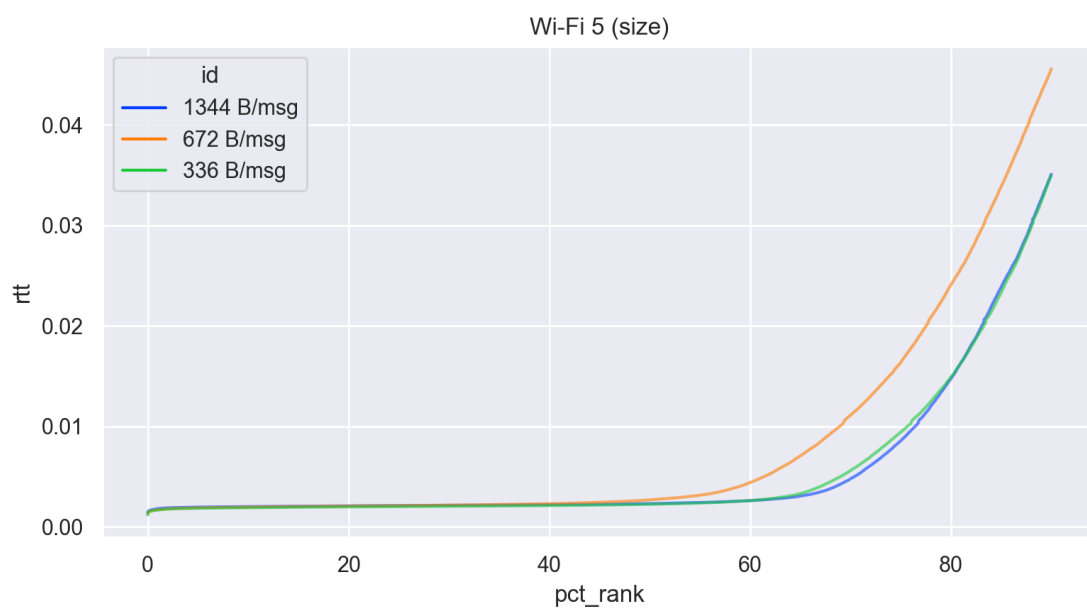


Figure A.3.14: Wi-Fi 5 RTT ranked (top 90%)

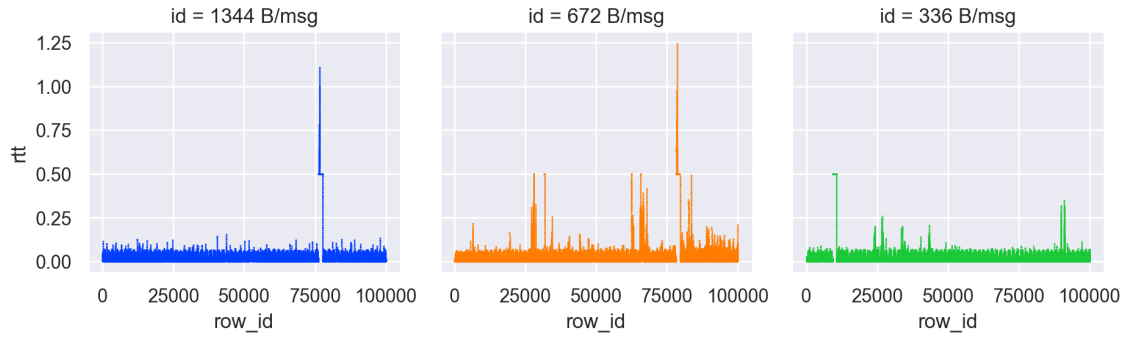


Figure A.3.15: Wi-Fi 5 message size scatterplots

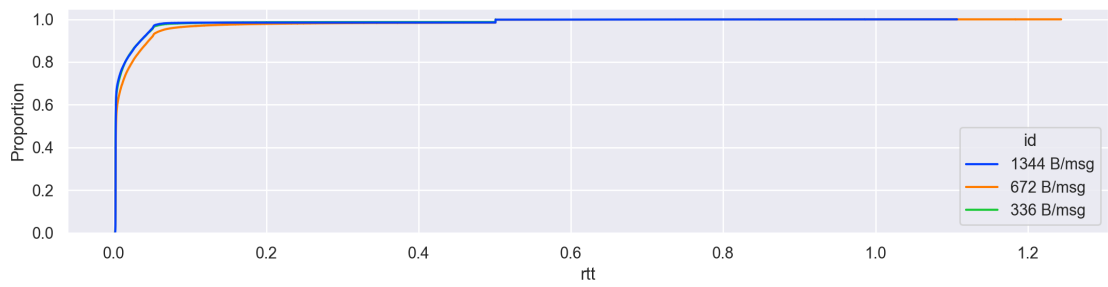


Figure A.3.16: Wi-Fi 5 message size ECDF

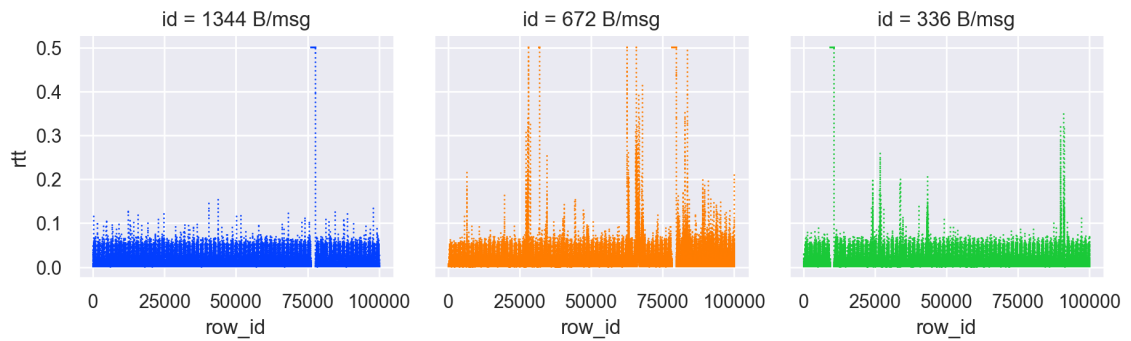


Figure A.3.17: Wi-Fi 5 message size scatterplots (top 99%)

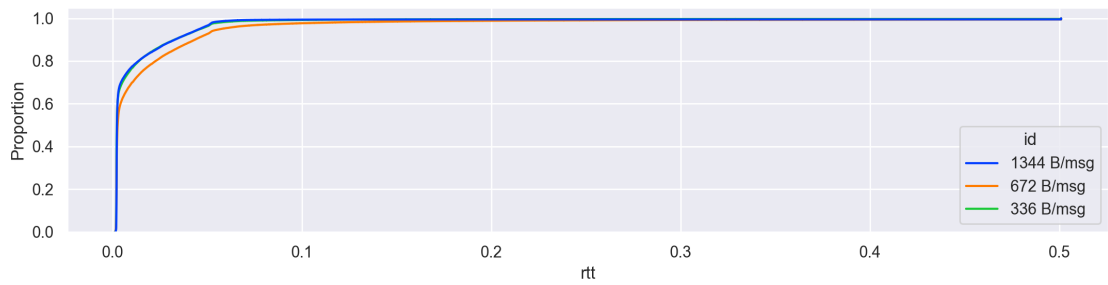


Figure A.3.18: Wi-Fi 5 message size ECDF (top 99%)

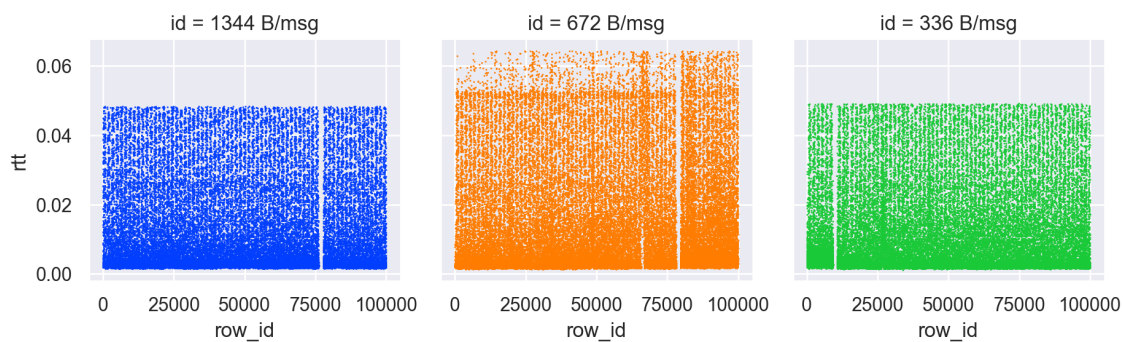


Figure A.3.19: Wi-Fi 5 message size scatterplots (top 95%)

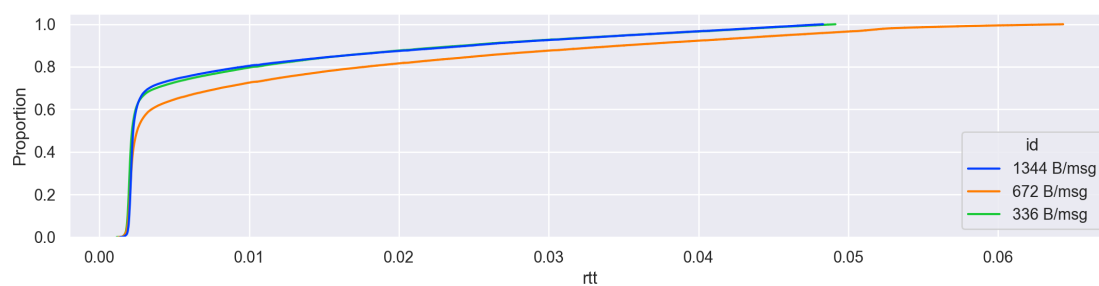


Figure A.3.20: Wi-Fi 5 message size ECDF (top 95%)

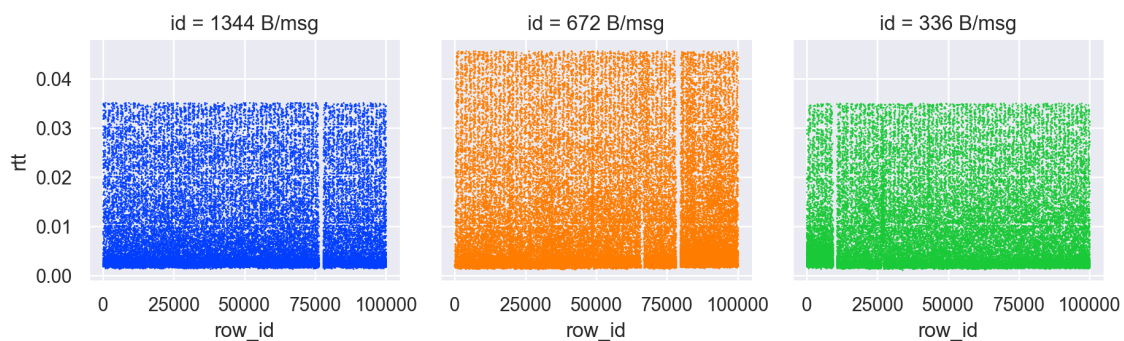


Figure A.3.21: Wi-Fi 5 message size scatterplots (top 90%)

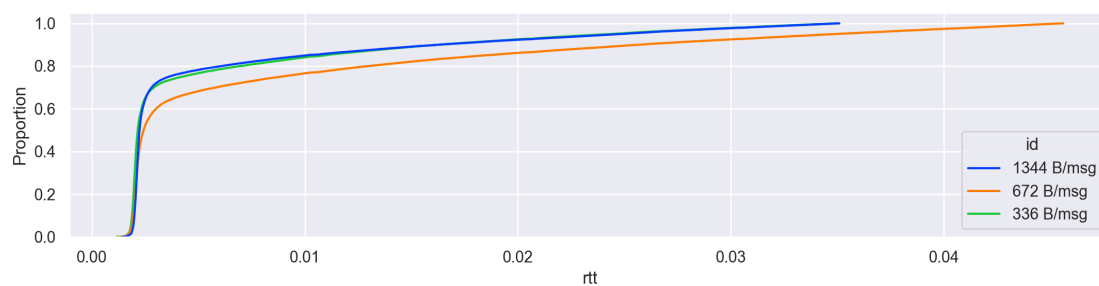


Figure A.3.22: Wi-Fi 5 message size ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
1344 B/msg	0.0169	0.0483	0.0074	0.0351	0.0055	0.0023
672 B/msg	0.0227	0.0644	0.0104	0.0456	0.0080	0.0027
336 B/msg	0.0162	0.0491	0.0074	0.0350	0.0055	0.0022

Table A.11: RTT statistics (Wi-Fi 5 | Size)

	max. pending	non-unique	mean jitter	max. jitter	timed out
1344 B/msg	0.5809	4612	0.0053	0.5609	1493
672 B/msg	0.7402	5288	0.0065	0.7302	1570
336 B/msg	0.5086	4471	0.0054	0.4986	1286

Table A.12: Delivery statistics (Wi-Fi 5 | Size)

A.4 Wi-Fi 6

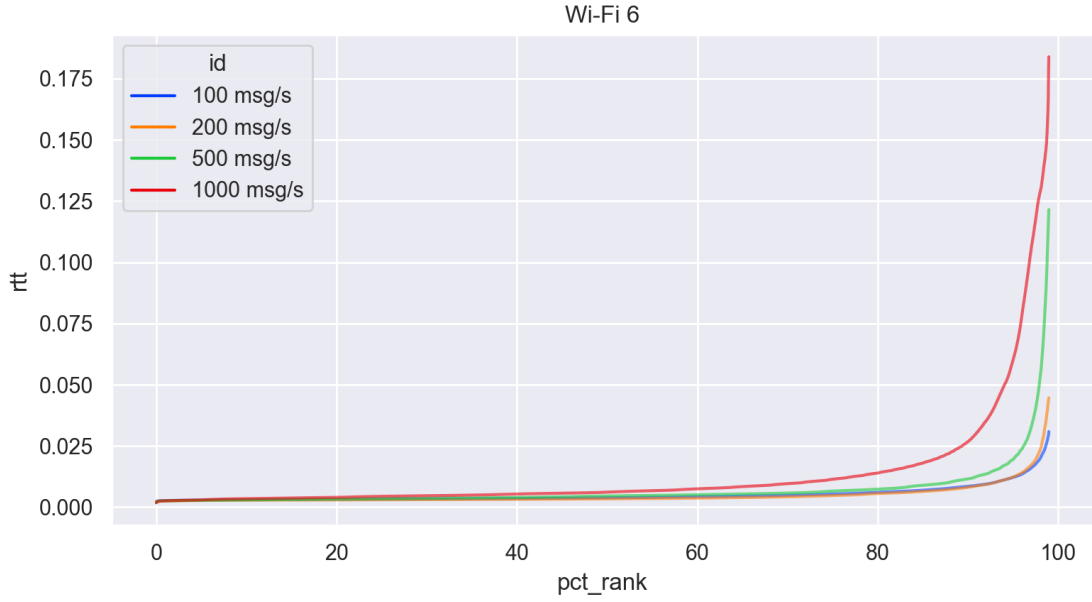


Figure A.4.1: Wi-Fi 6 (Rank to RTT, top 99%)

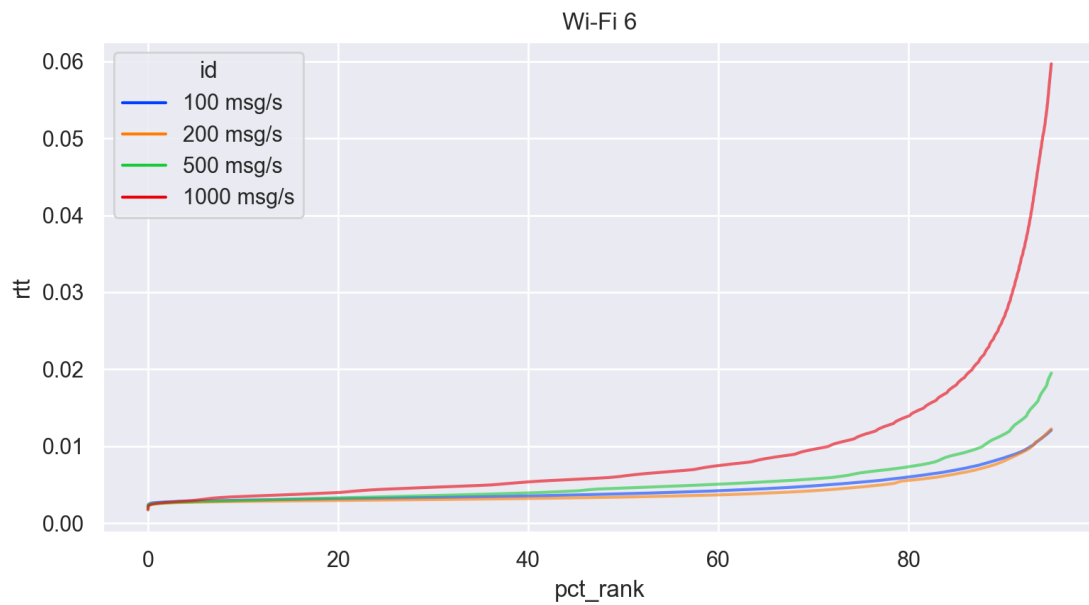


Figure A.4.2: Wi-Fi 6 (Rank to RTT, top 95%)

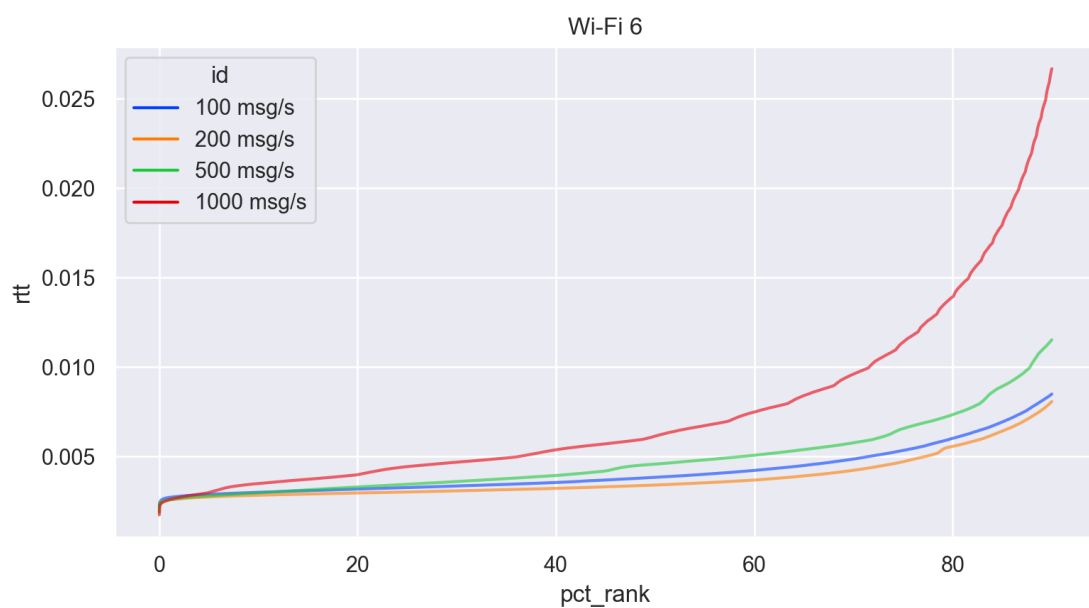


Figure A.4.3: Wi-Fi 6 (Rank to RTT, top 90%)

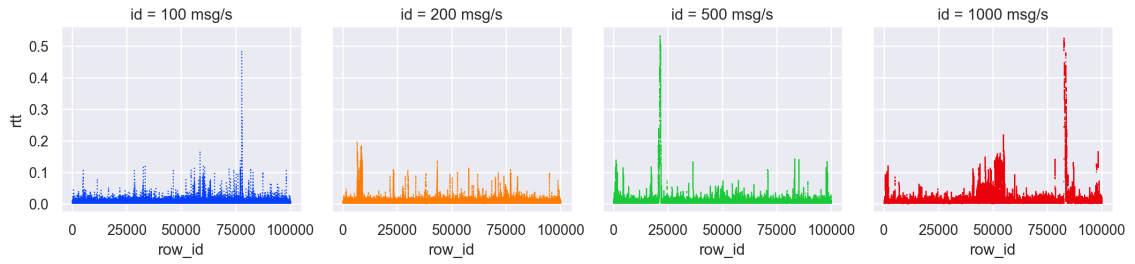


Figure A.4.4: Wi-Fi 6 scatterplots

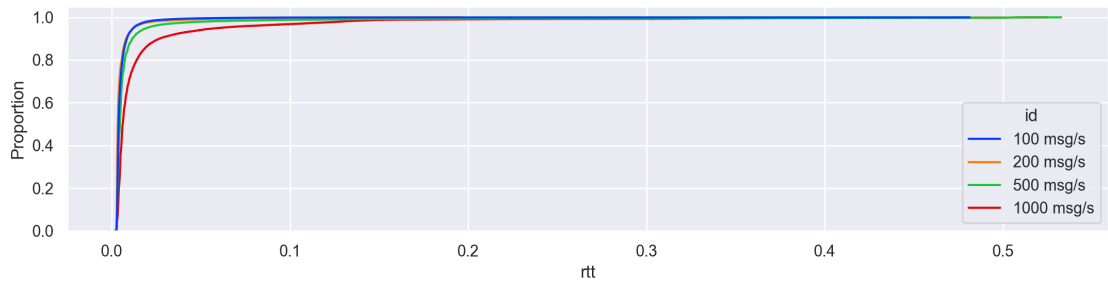


Figure A.4.5: Wi-Fi 6 ECDF

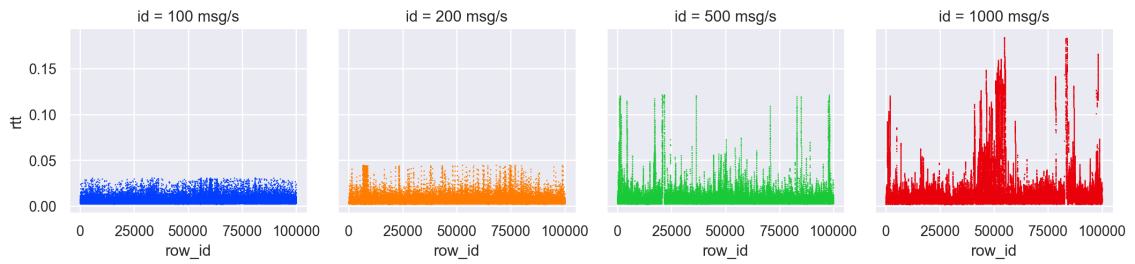


Figure A.4.6: Wi-Fi 6 scatterplots (top 99%)

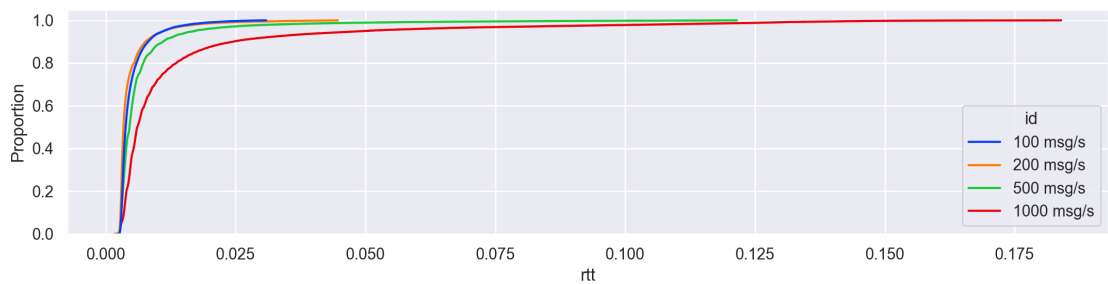


Figure A.4.7: Wi-Fi 6 ECDF (top 99%)

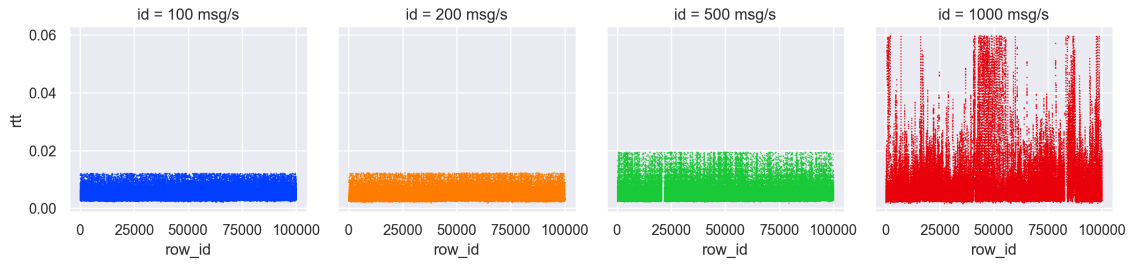


Figure A.4.8: Wi-Fi 6 scatterplots (top 95%)

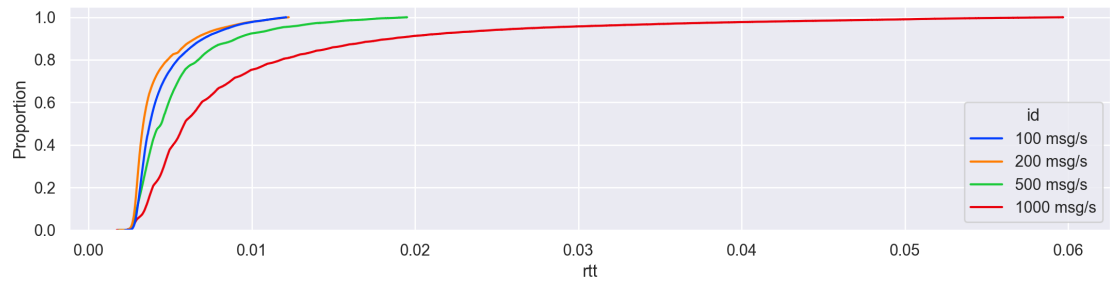


Figure A.4.9: Wi-Fi 6 ECDF (top 95%)

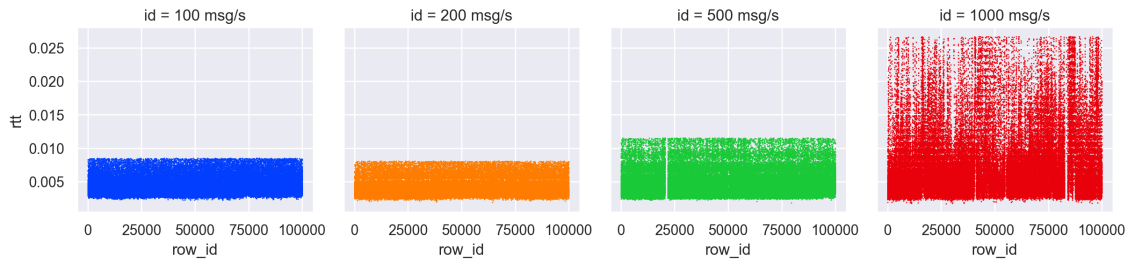


Figure A.4.10: Wi-Fi 6 scatterplots (top 90%)

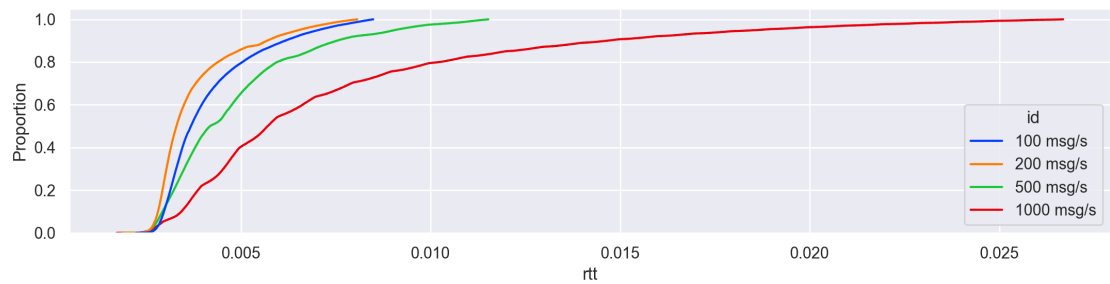


Figure A.4.11: Wi-Fi 6 ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
100 msg/s	0.0056	0.0121	0.0044	0.0085	0.0041	0.0038
200 msg/s	0.0055	0.0123	0.0041	0.0081	0.0038	0.0034
500 msg/s	0.0098	0.0195	0.0053	0.0115	0.0048	0.0046
1000 msg/s	0.0163	0.0597	0.0092	0.0267	0.0075	0.0061

Table A.13: RTT statistics (Wi-Fi 6)

	max. pending	non-unique	mean jitter	max. jitter	timed out
100 msg/s	0.4198	1075	0.0025	0.3798	0
200 msg/s	0.1654	2575	0.0018	0.1604	0
500 msg/s	0.1317	8479	0.0017	0.1263	192
1000 msg/s	0.2476	15168	0.0015	0.2468	134

Table A.14: Delivery statistics (Wi-Fi 6)

A.4.1 Different Payload Sizes

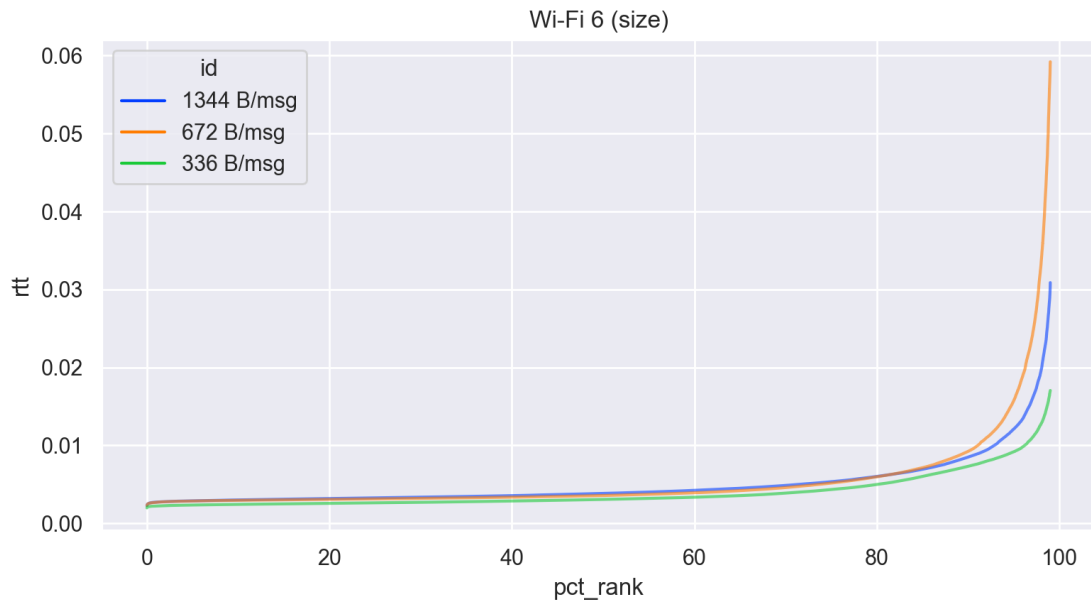


Figure A.4.12: Wi-Fi 6 RTT ranked (top 99%)

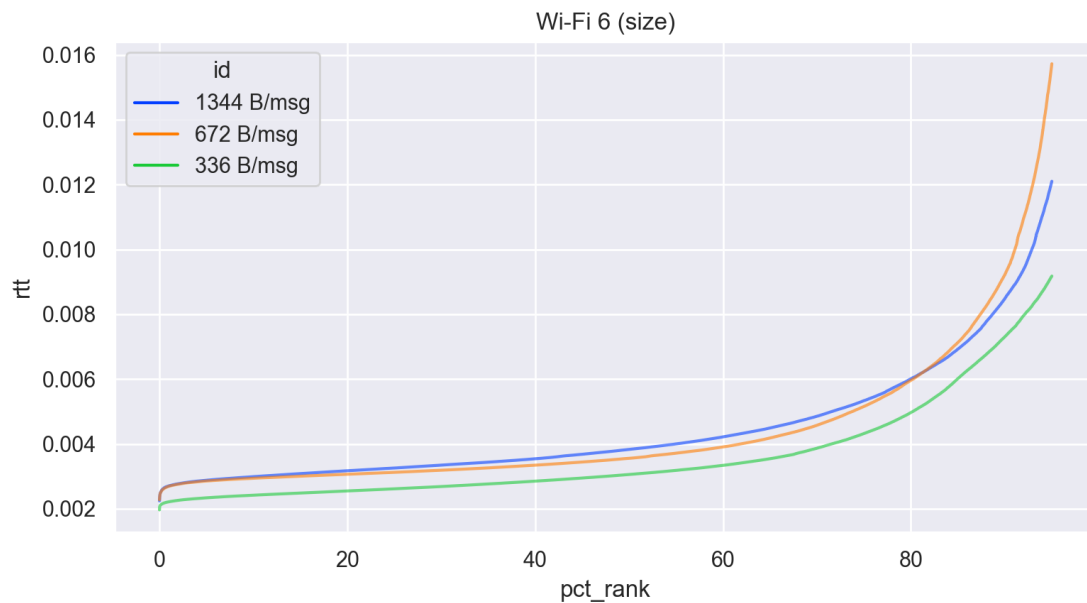


Figure A.4.13: Wi-Fi 6 RTT ranked (top 95%)

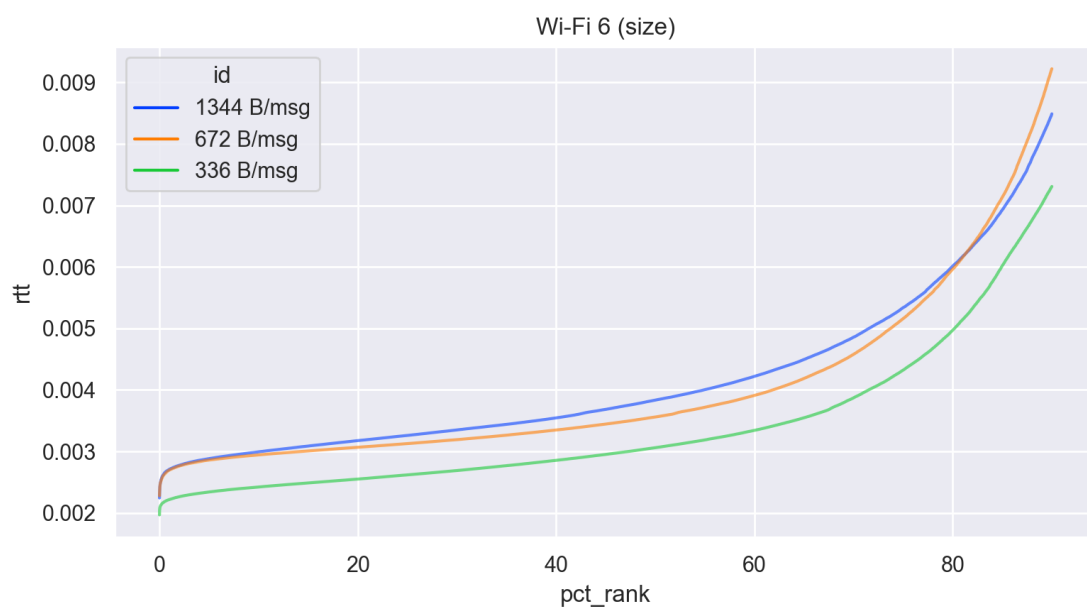


Figure A.4.14: Wi-Fi 6 RTT ranked (top 90%)

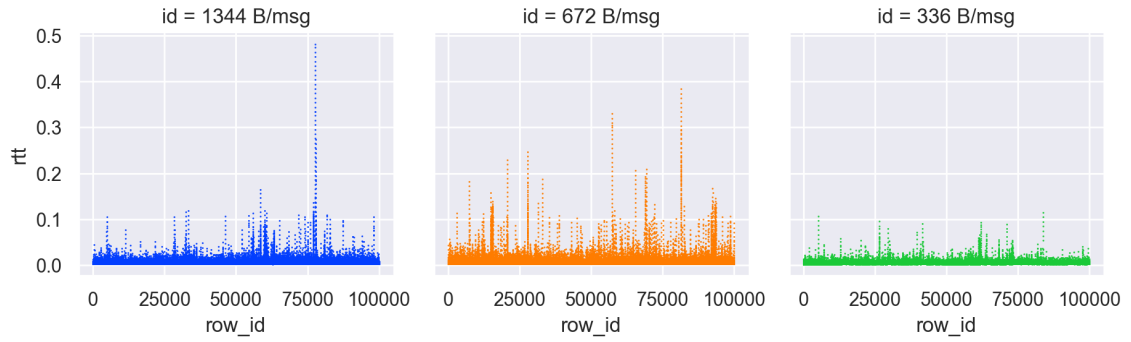


Figure A.4.15: Wi-Fi 6 message size scatterplots

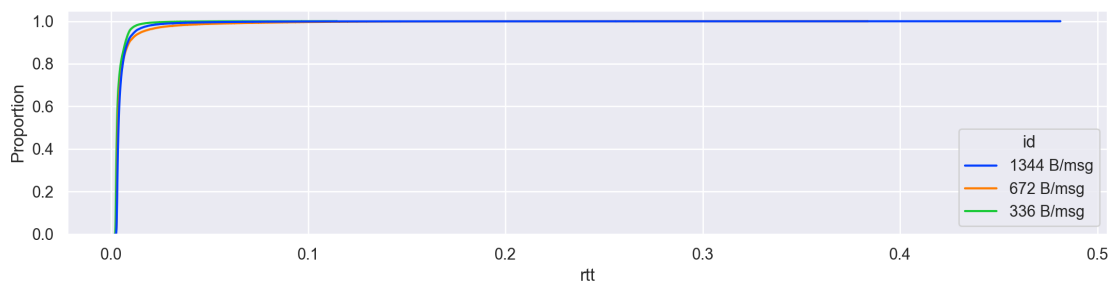


Figure A.4.16: Wi-Fi 6 message size ECDF

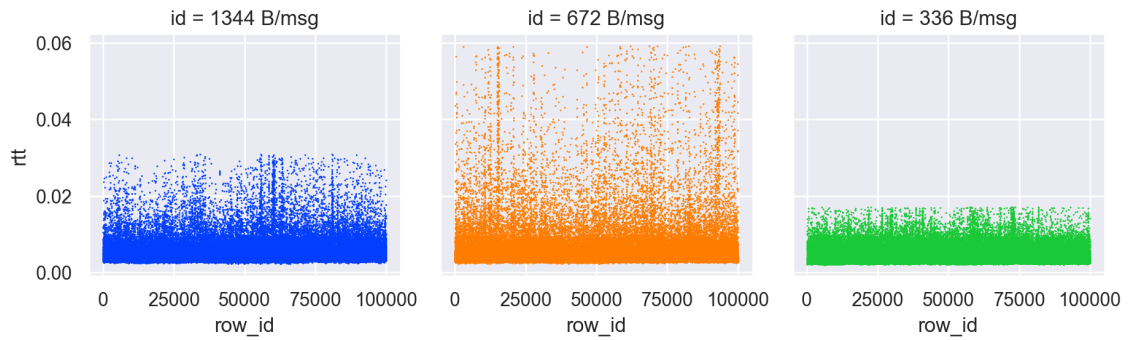


Figure A.4.17: Wi-Fi 6 message size scatterplots (top 99%)

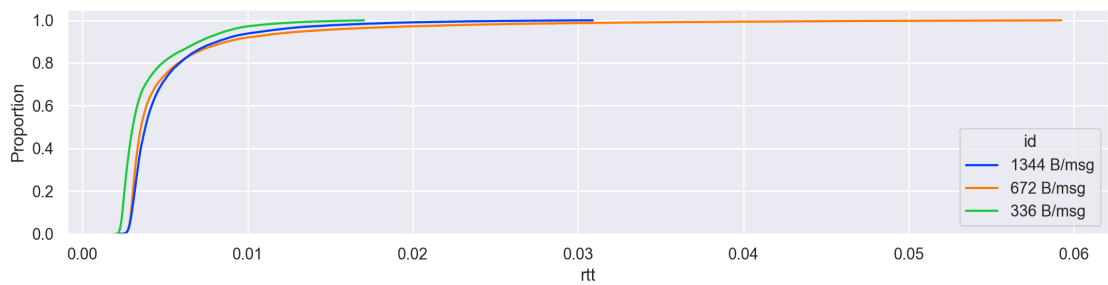


Figure A.4.18: Wi-Fi 6 message size ECDF (top 99%)

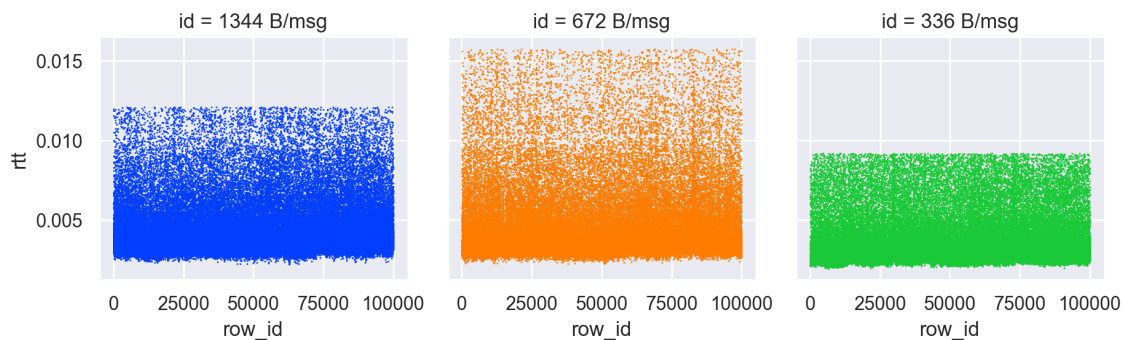


Figure A.4.19: Wi-Fi 6 message size scatterplots (top 95%)

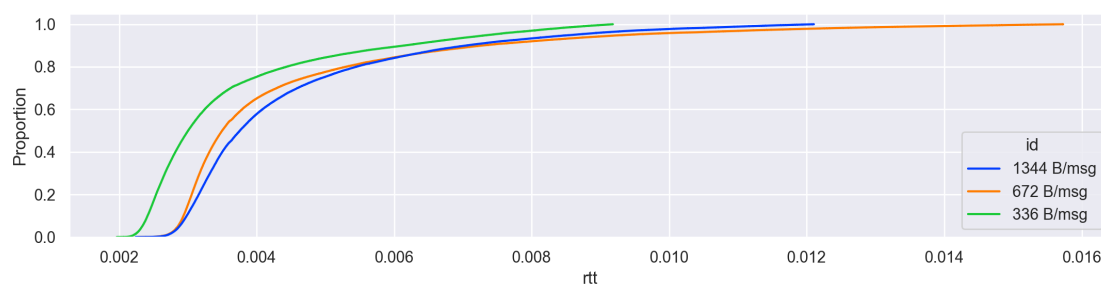


Figure A.4.20: Wi-Fi 6 message size ECDF (top 95%)

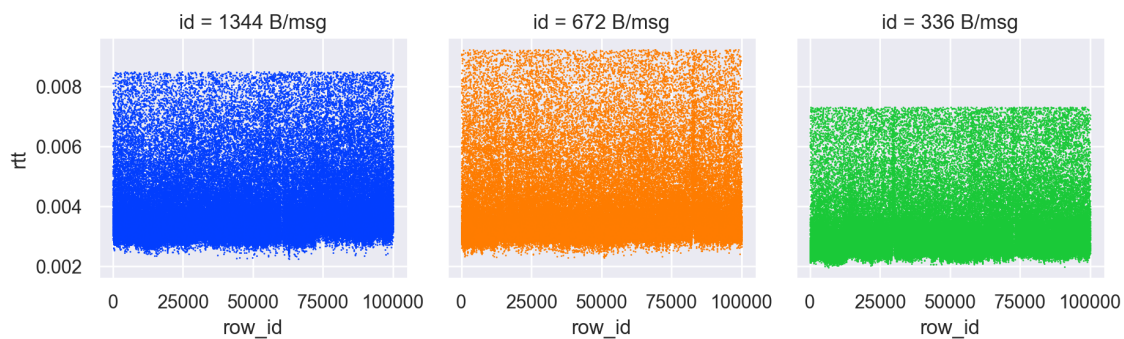


Figure A.4.21: Wi-Fi 6 message size scatterplots (top 90%)

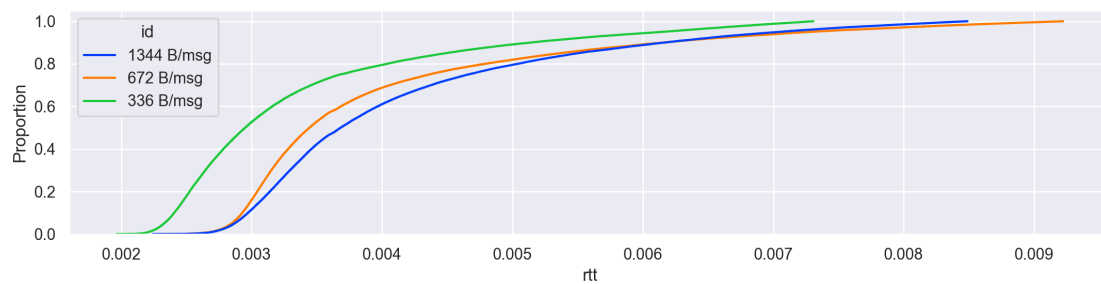


Figure A.4.22: Wi-Fi 6 message size ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
1344 B/msg	0.0056	0.0121	0.0044	0.0085	0.0041	0.0038
672 B/msg	0.0064	0.0157	0.0044	0.0092	0.0040	0.0036
336 B/msg	0.0042	0.0092	0.0036	0.0073	0.0034	0.0031

Table A.15: RTT statistics (Wi-Fi 6 | Size)

	max. pending	non-unique	mean jitter	max. jitter	timed out
1344 B/msg	0.4198	1075	0.0025	0.3798	0
672 B/msg	0.3354	1624	0.0027	0.3254	0
336 B/msg	0.1223	352	0.0020	0.1123	0

Table A.16: Delivery statistics (Wi-Fi 6 | Size)

A.5 Comparison of Technologies

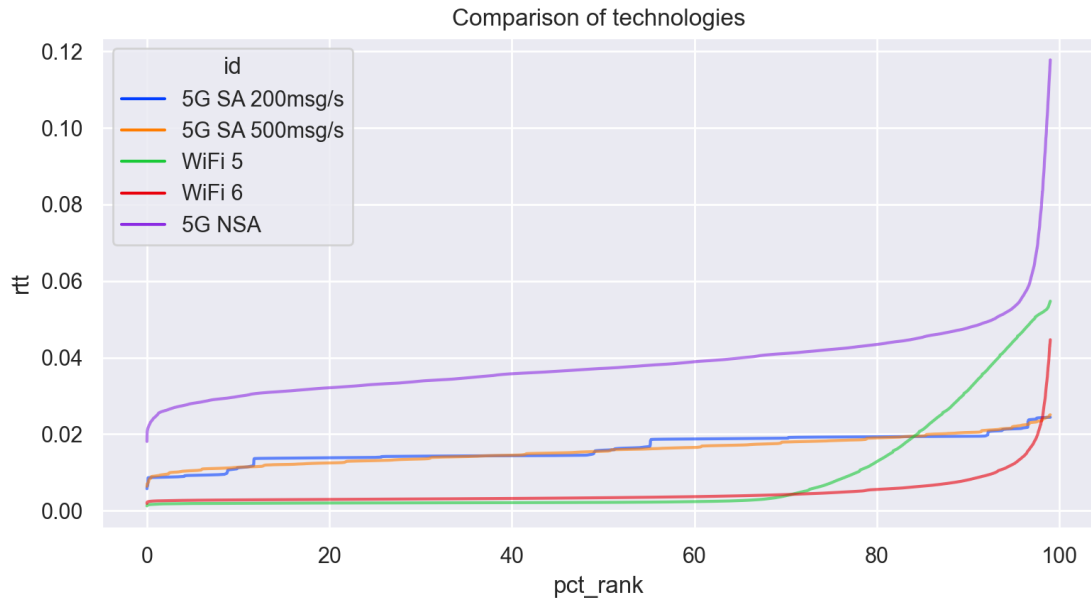


Figure A.5.1: All technologies compared (Rank to RTT, top 99%)

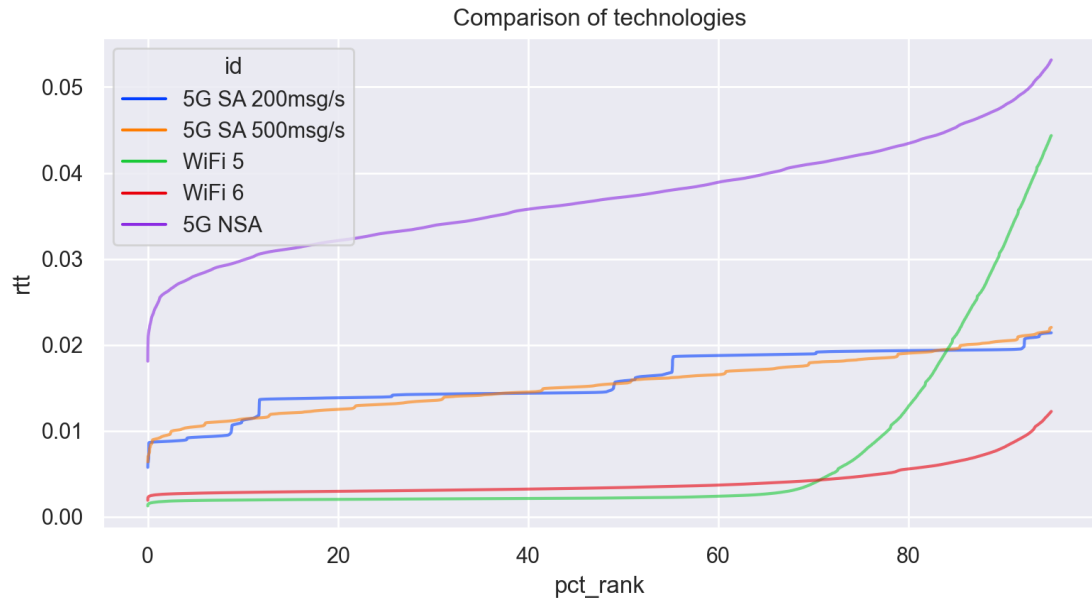


Figure A.5.2: All technologies compared(Rank to RTT, top 95%)

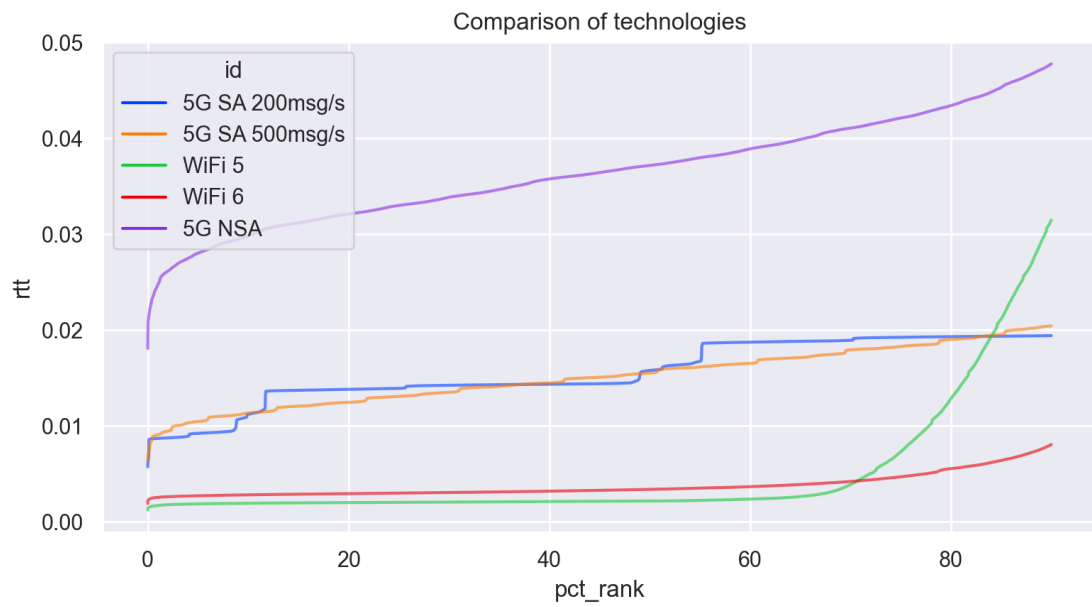


Figure A.5.3: All technologies compared (Rank to RTT, top 90%)

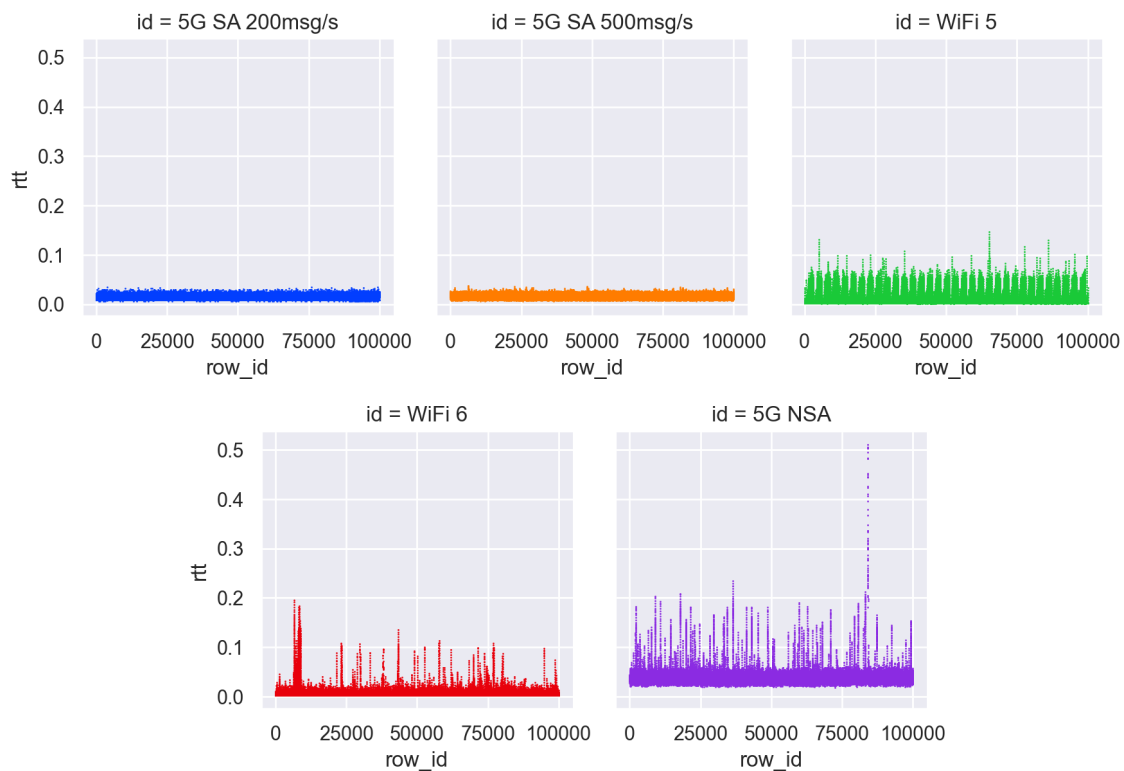


Figure A.5.4: All technologies compared scatterplots

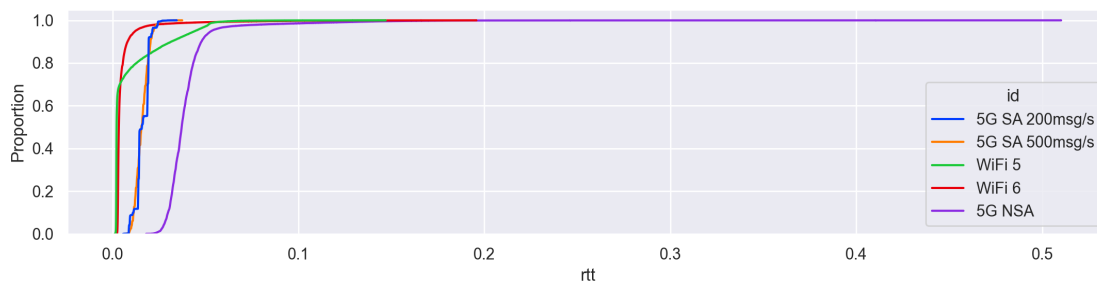


Figure A.5.5: All technologies compared ECDF

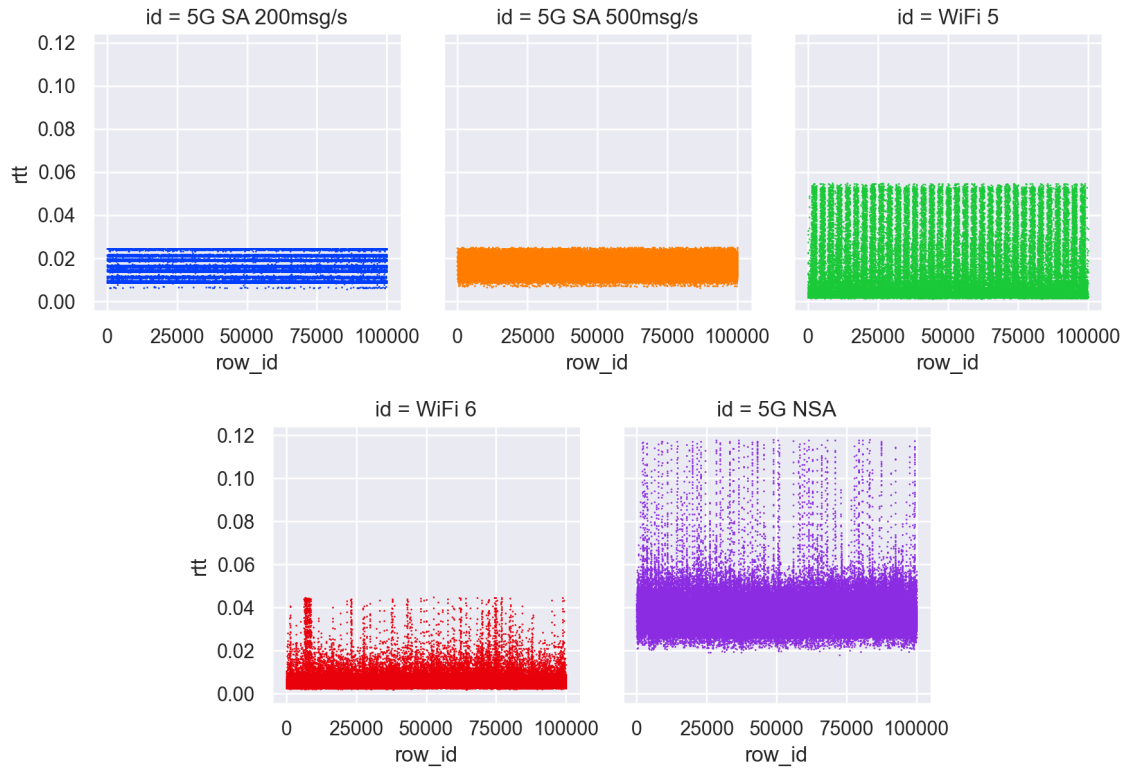


Figure A.5.6: All technologies compared scatterplots (top 99%)

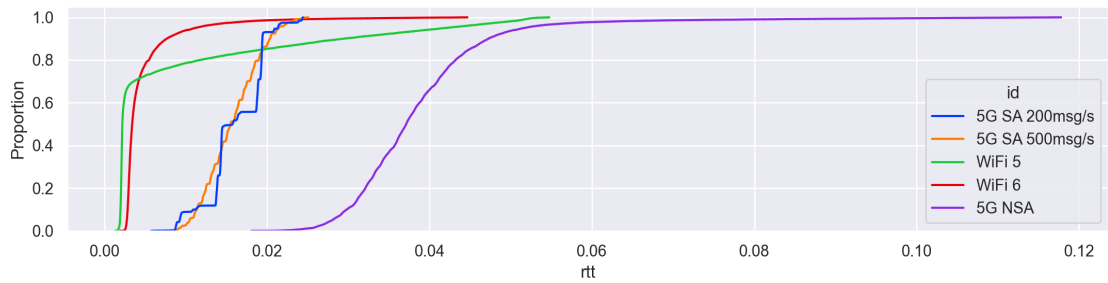


Figure A.5.7: All technologies compared ECDF (top 99%)

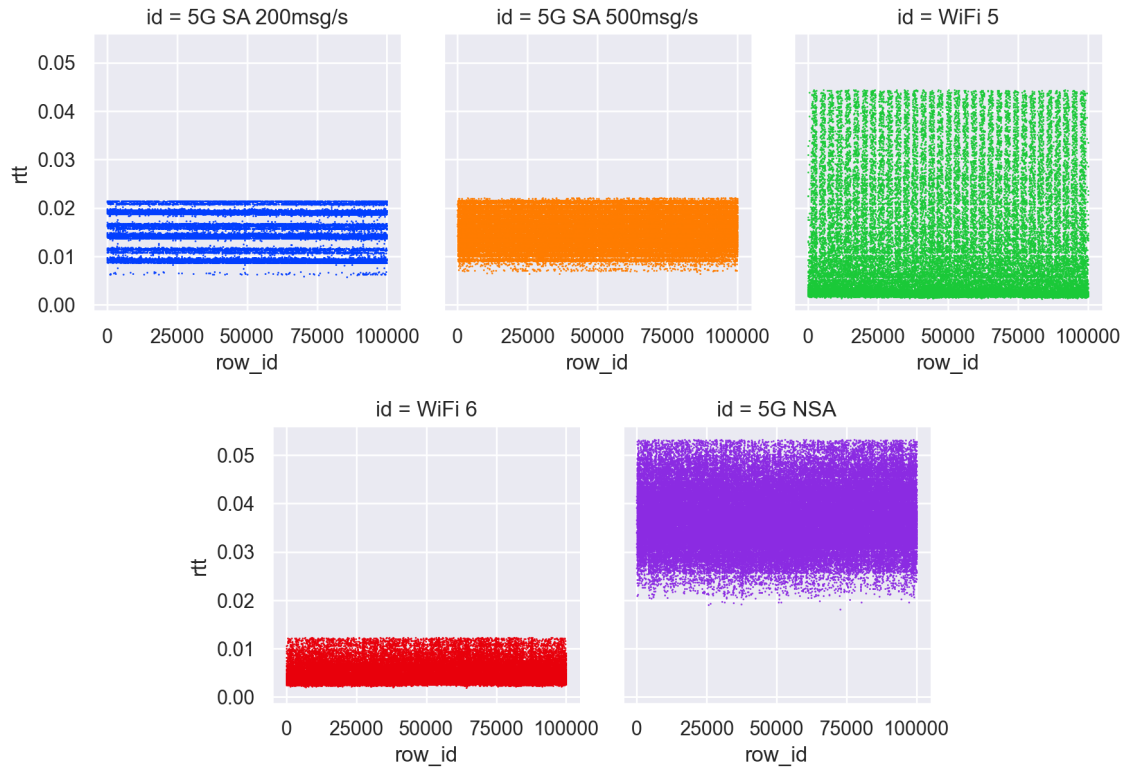


Figure A.5.8: All technologies compared scatterplots (top 95%)

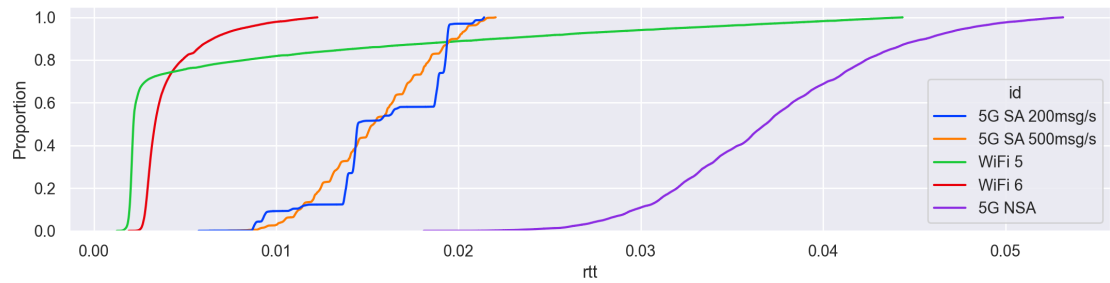


Figure A.5.9: All technologies compared ECDF (top 95%)

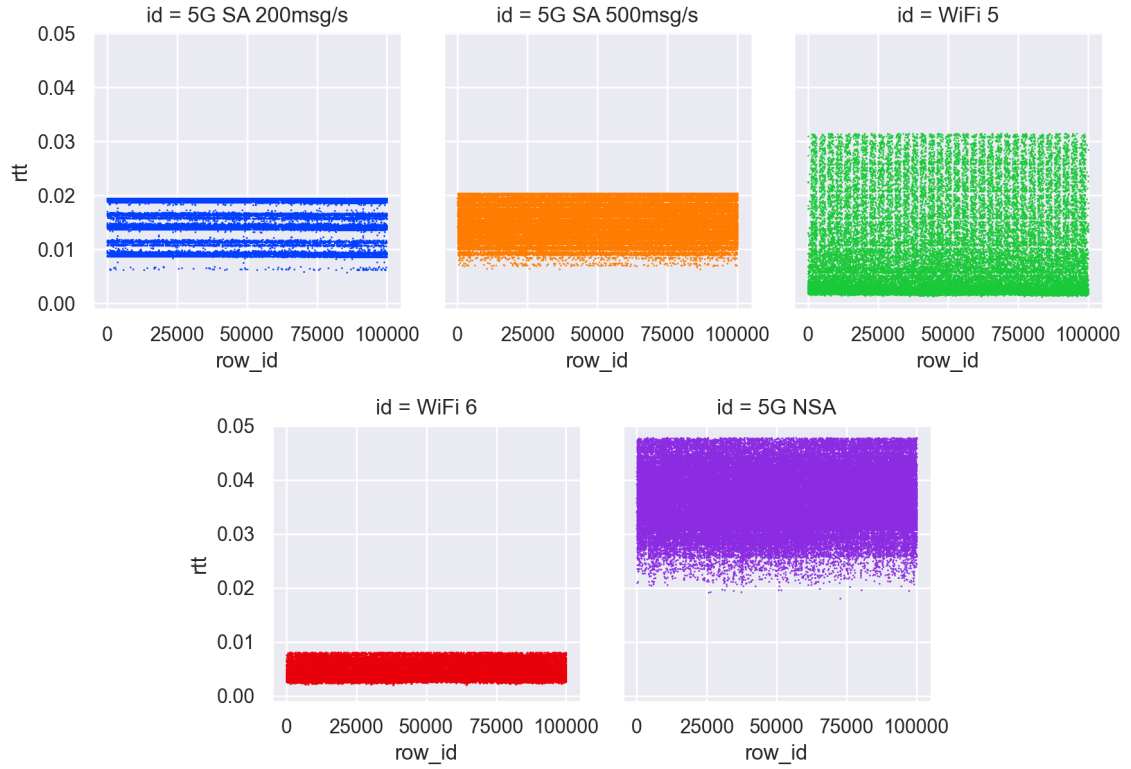


Figure A.5.10: All technologies compared scatterplots (top 90%)

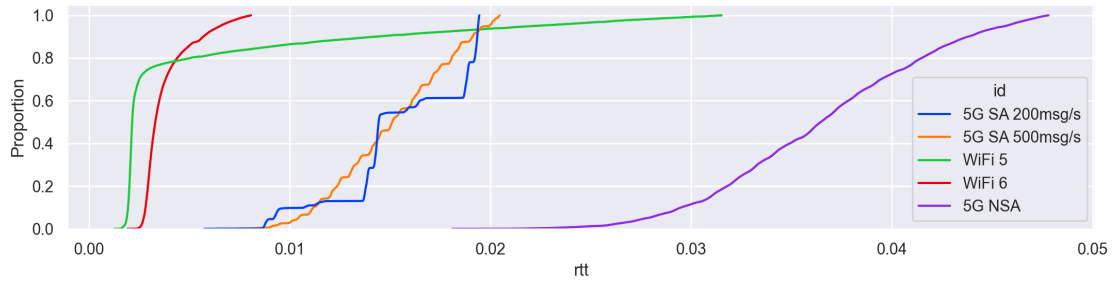


Figure A.5.11: All technologies compared ECDF (top 90%)

	mean rtt	p95	p95 mean	p90	p90 mean	median
5G SA ^{200msg/s}	0.0163	0.0214	0.0159	0.0194	0.0156	0.0158
5G SA ^{500msg/s}	0.0159	0.0220	0.0154	0.0205	0.0151	0.0155
WiFi 5	0.0090	0.0444	0.0067	0.0315	0.0049	0.0022
WiFi 6	0.0055	0.0123	0.0041	0.0081	0.0038	0.0034
5G (NSA)	0.0397	0.0532	0.0372	0.0478	0.0365	0.0372

Table A.17: RTT statistics (comparison of technologies)

	max. pending	non-unique	mean jitter	max. jitter	timed out
5G SA ^{200msg/s}	0.0298	12185	0.0048	0.0248	0
5G SA ^{500msg/s}	0.0259	9474	0.0030	0.0239	0
WiFi 5	0.1336	4383	0.0030	0.1286	0
WiFi 6	0.1654	2575	0.0018	0.1604	0
5G NSA	0.1497	15835	0.0047	0.1414	4

Table A.18: Delivery statistics (comparison of technologies)