Congestion Control Algorithms for the Constrained Application Protocol (CoAP)

Iivo Raitahila

Helsinki May 14, 2019

UNIVERSITY OF HELSINKI

Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Studieprogram — Study Programme Computer Science

Tekijä — Författare — Author
Iivo Raitahila

Työn nimi — Arbetets titel — Title

Congestion Control Algorithms for the Constrained Application Protocol (CoAP)

Ohjaajat — Handledare — Supervisors
Markku Kojo

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year May 14, 2019	Sivumäärä — Sidoantal — Number of pages 59 pages

Tiivistelmä — Referat — Abstract

The Internet of Things (IoT) consists of physical devices, such as temperature sensors and lights, that are connected to the Internet. The devices are typically battery powered and are constrained by their low processing power, memory and low bitrate wireless communication links. The vast amount of IoT devices can cause heavy congestion in the Internet if congestion is not properly addressed.

The Constrained Application Protocol (CoAP) is an HTTP-like protocol for constrained devices built on top of UDP. CoAP includes a simple congestion control algorithm (DefaultCoAP). CoAP Simple Congestion Control/Advanced (CoCoA) is a more sophisticated alternative for DefaultCoAP. CoAP can also be run over TCP with TCP's congestion control mechanisms.

The focus of this thesis is to study CoAP's congestion control. Shortcomings of DefaultCoAP and CoCoA are identified using empirical performance evaluations conducted in an emulated IoT environment.

In a scenario with hundreds of clients and a large buffer in the bottleneck router, DefaultCoAP does not adapt to the long queuing delay. In a similar scenario where short-lived clients exchange only a small amount of messages, CoCoA clients are unable to sample a round-trip delay time. Both of these situations are severe enough to cause a congestion collapse, where most of the link bandwidth is wasted on unnecessary retransmissions.

A new retransmission timeout and congestion control algorithm called Fast-Slow Retransmission Timeout (FASOR) is congestion safe in these two scenarios and is even able to outperform CoAP over TCP. FASOR with accurate round-trip delay samples is able to outperform basic FASOR in the challenging and realistic scenario with short-lived clients and an error-prone link.

ACM Computing Classification System (CCS):
Networks $\sim$ Network protocol design,
Networks $\sim$ Network performance evaluation

Avainsanat — Nyckelord — Keywords
CoAP, CoCoA, FASOR, congestion control, IoT protocols

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

The idea of the Internet of Things (IoT) is to connect physical devices to the Internet [AFGM$^+$15]. Such devices include temperature sensors, thermostats, lightbulbs, GPS-trackers and irrigators, for example. The sensors can be read and actuators can be controlled over the Internet. It is expected that the amount of IoT entities reaches 212 billion by the end of 2020 and that machine-to-machine traffic constitutes up to 45% of the Internet traffic by 2022 [AFGM$^+$15].

These devices typically have low processing power, low amounts of memory and operate on battery power. Smartphones and single board computers, such as Raspberry Pi and Arduino computers, fitted with sensors and a wired network connection, are examples of more powerful IoT devices.

IoT devices can use different communication technologies, such as WiFi [Wi-Fi], Bluetooth [Bluetooth], Z-wave [Z-Wave], LTE-Advanced [GRM$^+$10] and NFC [NFC]. The communication links are typically wireless, have a low bitrate, and are prone to a high packet-error rate [SHB14].

Special embedded Operating Systems include TinyOS [LMP$^+$05] and Contiki [DGV04], but also Android can be used. Abstraction layers that ease network programming can take a large amount of the sparse resources [AFGM$^+$15].

The Constrained Application Protocol (CoAP) [SHB14] is an HTTP-like protocol for constrained devices. The CoAP protocol tackles these issues by having low overhead and simple algorithms. To lower the overhead, the protocol is implemented on top of UDP by default and optional functionality for reliable message delivery is included with necessary congestion control.

CoAP includes a simple congestion control algorithm (DefaultCoAP) that can be replaced with other algorithms. The hypothesis is that the more complex algorithms provide better performance at least in certain cases. Such algorithms include CoAP Simple Congestion Control/Advanced (CoCoA) [BBGD17], a congestion control algorithm specially meant for CoAP. CoAP can also be used over TCP [BLT$^+$18] using TCP's congestion control algorithms.

The research problem of this thesis is to study CoAP's congestion control. This is important because communication occurs usually over low-bitrate bottleneck links and a potentially vast number of IoT devices can cause heavy congestion if congestion is not properly addressed.

This thesis includes empirical performance evaluation of DefaultCoAP, CoCoA and CoAP over TCP. The evaluation is conducted in a network environment that emulates an IoT environment, where multiple clients communicate with one server over a constrained and error-prone bottleneck link. The algorithms are implemented into libcoap, an open-source CoAP library for C [libcoap]. Both ends of the test network (the clients and the server) use the library and actual implementations of the algorithms. A new retransmission timeout (RTO) and congestion control algorithm called Fast-Slow Retransmission Timeout (FASOR) [JKRC18] addresses the shortcomings that were found.

The rest of the thesis is organized as follows. Section 2 describes the general setting of IoT communications, including descriptions of IoT devices, IoT networks, congestion and the Constrained Application Protocol (CoAP). Section 3 covers the existing congestion control algorithms (DefaultCoAP, TCP RTO and CoCoA) while Section 4 contains performance evaluation results from other publications. The test arrangements for the new performance evaluations are described in Section 5 and results for the existing congestion control algorithms are discussed in Section 6. Section 7 introduces the new FASOR algorithm and its results are compared with the baseline algorithms in Section 8. Sections 9 and 10 review use cases for the congestion control algorithms, future work and conclude the thesis.

2 Communications in IoT

This section discusses the characteristics of IoT devices and networks, protocols used in IoT environments and congestion control in general. Running CoAP over TCP is not a part of the basic CoAP specification, but is a separate standard and discussed in its own subsection. At the end of the Chapter, an overview figure of an IoT network is presented.

2.1 Constrained devices and networks

IoT devices can be constrained in many aspects. Constrained devices (nodes) do not have the same characteristics that are taken for granted for regular devices connected to the Internet [BEK14]. There are, for example, constraints in physical size and weight, manufacturing cost and available power. Energy and network usage optimizations are a design priority, as the devices can be battery powered and wireless transmission consumes a lot of energy. Using constrained devices in a network causes constraints to the network, too.

The RFC 7228 defines classes for constrained devices. A class 0 device has less than 10 KiB of RAM and less than 100 KiB of storage capacity. Class 1 and 2 devices have about 10 KiB and 50 KiB of RAM, respectively, and about 100 KiB and 250 KiB of storage capacity, respectively. Class 0 devices are most likely too constrained to communicate securely when connected directly to the Internet, requiring the use of a proxy or a gateway. Class 1 devices are capable of using a special constrained protocol stack such as CoAP. Class 2 devices can use regular protocol stacks but benefit from lightweight protocols [BEK14].

A network can be constrained independently from the devices. Again, these networks do not have the same link-layer characteristics that are taken for granted with common networks in the Internet. Typical constraints are low bitrate and high packet-error rate. The communication links are typically wireless [SHB14], and errors may occur in bursts [DMK+01]. Even wired media can be error prone, such as when communicating over power lines [GAMC18].

Because the devices can be battery powered, low power usage is often more important than for example high bandwidth and range. Low data rate services account for more than 67% of total IoT services [CMHH17]. Video surveillance is one example of a high data rate service, requiring for example LTE-Advanced or WiFi connection.

IoT devices can use different communication protocols that have different properties [CMHH17]. For example WiFi, Bluetooth and Z-Wave are short-distance protocols. Low-power wide area networks (LPWAN) are suitable for long-distance IoT communication [AVTP+17]. Typically higher data rates cause higher energy consumption and shorter range. LoRaWAN [LoRaWAN] is a LPWAN technology that provides a data rate of 0.3 - 50 kbit/s, a battery lifetime of around 10 years and a communication range of 2-15 kilometers. LoRaWAN devices communicate with a gateway that connects them to, for example, the Internet. Other LPWAN technologies include SigFox [Sigfox] with 100 bit/s uplink, Ingenu [Ingenu] with 624 kbit/s uplink and 156 kbit/s downlink, and Weightless-W [Weightless] with up to 1 Mbit/s data rate. LoRaWAN, among others, favours uplink communication from the devices to the server/gateway.

The 3rd Generation Partnership Project (3GPP) created multiple cellular standards for IoT that operate in mobile operators' networks instead of using private access points/gateways [AVTP+17]. Roaming is one advantage of using cellular networks [CMHH17]. These standards include the GSM compatible EC-GSM-IoT [EC-GSM], LTE compatible eMTC [eMTC-NB] and new Narrow Band IoT (NB-IoT) [eMTC-NB]. NB-IoT has a data rate of up to 250 kbit/s uplink and downlink. In a scenario where the carrier has reserved 200 kHz bandwidth for NB-IoT, the uplink peak rate is 66.7 kbit/s and 32.4 kbit/s for downlink [CMHH17]. In this scenario the battery should last about 10 years when a 200-byte message is sent once a day.

2.2 The Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) [SHB14] is a web transfer protocol designed for machine-to-machine applications that use constrained nodes. To increase packet delivery probability and decrease the need for packet fragmentation, the message overhead is kept as small as possible. The protocol is similar to HTTP in many ways (method codes, request methods such as GET and POST, URIs). It uses the REST architecture with a client/server model, where the client sends requests to the server and the server sends responses.

CoAP is built on top of UDP and provides optional reliability in the form of confirmable messages. Confirmable messages require acknowledgements and are retransmitted if not acknowledged using a simple RTO-based stop-and-wait mechanism. The acknowledgement may contain the response piggybacked or the response can arrive later as a separate message if the processing takes too long .

For congestion control CoAP uses a simple binary exponential scheme. The initial RTO is a random value between 2 and 3 seconds, and the current RTO value is doubled on timer expiration. By default the maximum retransmission attempts is 4 and only one outstanding interaction per server is allowed. These parameters, among others, can be configured.

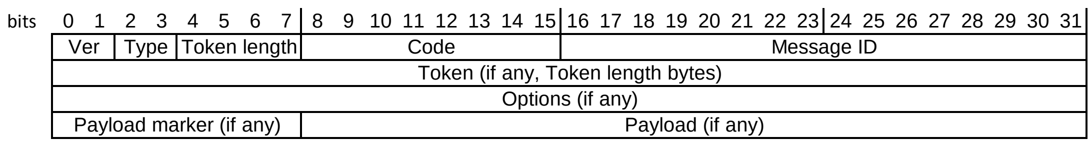

Figure 1: CoAP message structure, adapted from [SHB14]

The header length is fixed to 4 bytes, optionally followed by a token, options and a payload (see Figure 1). The header contains a message ID that is used for duplicate detection and reliability (the acknowledgement message contains the same message

ID as the confirmable message). Non-confirmable messages use the message ID for duplicate detection only. In case the response is not piggybacked, an empty acknowledgement message with the same message ID is returned. The separate response message is associated to the request using a Token. The token is a client generated value between 0 and 8 bytes long that is echoed back by the server. An empty token value can be used when piggybacked responses are used.

Because CoAP is based on UDP, which supports only fragmentation, there is a limit on how large payloads can be. To avoid IP fragmentation, the payload size should be maximum 1024 bytes (1280 bytes IPv6 MTU, can be less for constrained networks). Larger payloads should use block-wise transfers [BS16]. For example, firmware updates do not fit in a single UDP datagram. In block-wise transfers the transferred resource is split into multiple 16 to 1024 byte pieces that are transferred in multiple ordinary CoAP request-response pairs in a stop-and-wait fashion. Additional block options are used, where for example the block size is negotiated. Normal congestion control rules apply. If a block is lost, it can be easily re-requested individually. The transfer is mostly stateless.

A CoAP proxy is a CoAP client that can perform requests on some other client's behalf, for example to reduce direct communication with a battery powered CoAP endpoint [SHB14]. A proxy can be a forward-proxy, allowing communication from the CoAP devices, or a reverse proxy, allowing external communication to the CoAP devices. A proxy can also translate between protocols, such as HTTP as defined in RFC 7252. A CoAP-HTTP proxy allows CoAP clients to use HTTP resources and a HTTP-CoAP proxy allows the use of CoAP resources from web interfaces, for example.

The CoAP protocol contains more features, such as security using Datagram Transport Layer Security (DTLS), that are not discussed in this thesis to focus on the topic. Unless explicitly specified, all messages discussed in this thesis are confirmable messages .

2.3 CoAP over TCP

The CoAP protocol exchanges messages over UDP by default [SHB14], but also TCP can be used. RFC 8323 [BLT+18] specifies the use of CoAP over TCP, TLS and WebSockets. CoAP over UDP clients in a network can also communicate with a gateway that connects to another network using CoAP over TCP. Transport Layer

Security (TLS) provides security for CoAP over TCP in a similar manner as DTLS provides it for CoAP over UDP, and WebSockets can be used in CoAP applications running in a web browser that cannot use other protocols than HTTP. These latter two techniques are not discussed further in this thesis.

TCP is a connection-oriented protocol that provides reliable transmission and ordered data delivery, used for example in WWW and email [GAMC18]. In IoT communications UDP is often preferred over TCP, because TCP's connection establishment increases delay, the TCP header is larger than the UDP header, and TCP does not support multicast nor unreliable communication.

Some other drawbacks of TCP are erroneously criticized [GAMC18]. The congestion control was designed for wired links where packet losses are assumed to be caused by congestion and not link errors that are often present in IoT networks. The problem of distinguishing congestion from link errors is not present only in TCP, but also in other Automatic Repeat reQuest (ARQ) based protocols such as CoAP. The delay caused by connection establishment is a smaller issue if the connection is long lived or TCP Fast Open [CCRJ14] is used, where data can be embedded already in the SYN and SYNACK packets. TCP is also regarded as a complex protocol and while it is more complex than UDP, it was designed for computers from the eighties that resemble current constrained devices.

UDP is the recommended transport for CoAP, but some networks block UDP traffic. These networks include enterprise networks and geographically remote networks, with 2 to 4 percent of terrestrial access networks blocking UDP traffic and 0.3 percent rate-limit UDP traffic [BLT+18]. TCP connections have longer NAT binding timeouts (mean 386 minutes vs 160 seconds for TCP and UDP respectively [BLT+18]), requiring less keepalive messages.

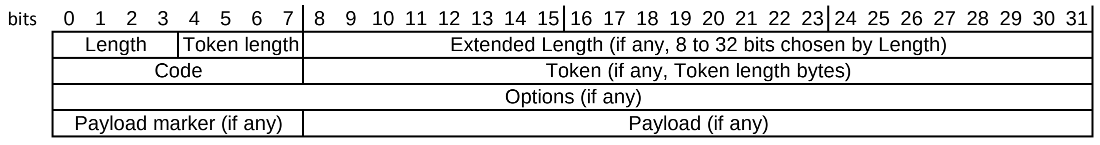

Figure 2: CoAP over TCP message structure, adapted from [BLT+18]

The CoAP over TCP header is different from the regular CoAP header (see Figure 2). The reliability and deduplication features of CoAP over UDP are redundant when CoAP is used over TCP, thus the Type and Message ID fields are removed. The type of the message will always be similar to confirmable, because TCP is a reliable transport protocol. The Version field is removed, and if required, version negotiation can be achieved with the Capabilities and Settings Messages (CSM). As TCP is a stream oriented protocol, the length of the message is a necessary piece of information and it is added to the CoAP header.

After the TCP three-way handshake, both endpoints exchange Capabilities and Settings Messages (CSM) that contain settings such as the maximum message size. The message can be empty if default settings are used. The connection initiator does not have to wait for the recipient's CSM message and may send other messages right after sending the CSM to reduce delay. Due to Nagle's algorithm [Nag84] there might be an additional delay of one RTT before the sender can send a message after the CSM message. Nagle's algorithm prevents the transmission of small packets when there is unacknowledged data to reduce overhead caused by the IP and TCP headers. When Nagle's algorithm is used, the CSM message is sent, but the next small message is queued and waits for an acknowledgement for the CSM message.

It is not mandatory to implement all TCP features [GCS19, GAMC18]. Such a lightweight implementation is TCP standard compliant and may have, for example, lower data throughput, but is more suitable for IoT devices. These tactics include setting the maximum segment size (MSS) into a fixed low value (1220 bytes with IPv6) for avoiding Path MTU discovery, and advertising a TCP window size of one MSS to simplify congestion and flow control. With one MSS window size options such as Window scale can be left unimplemented, because the window size is so small that scaling is unnecessary. On the other hand, sophisticated features such as Explicit Congestion Notification (ECN) may be beneficial in constrained networks to notify about congestion without packet drops and expensive retransmissions. uIP [Dun03] is an implementation of a TCP/IP stack for constrained devices that has a code size of about 5 kilobytes.

2.4 Other IoT protocols

CoAP is not the only application/transport protocol used in IoT. Other protocols include Message Queue Telemetry Transport (MQTT) [MQTT], Extensible Messaging and Presence Protocol (XMPP) [XMPP], Advanced Message Queuing Protocol (AMQP) [AMQP] and Data Distribution Service (DDS) [DDS]. Less constrained devices can use common protocols such as HTTP. Some are more suitable for different scenarios and environments than others [AFGM+15].

MQTT consists of publishers, subscribers and brokers. Subscribers register their interest to specific topics to a broker. Publishers send messages with topics to the broker, which then relays the messages to interested subscribers.

XMPP was created for instant messaging (chatting). Clients connect to a server and servers can be connected to each other in a decentralized fashion. Servers can be gateways to other networks, such as the Internet. The communication is based on XML, which adds overhead compared to binary communication.

AMQP supports point-to-point and publish/subscribe communication with at-most-once, at-least-once and exactly once delivery guarantees. A broker, similar to MQTT's, stores messages in queues for sending to subscribers.

DDS is a publish/subscribe protocol without brokers. Instead, the messages are sent using multicasting.

Out of the aforementioned protocols only CoAP supports RESTful communication. The request/response method is supported by XMPP in addition to CoAP. The others use a publish/subscribe method, which is an option for XMPP and also proposed for CoAP [KKJ19]. The CoAP Observe feature [Har15] is similar to publish/subscribe, in which the server sends updates to clients when a resource they have subscribed to is modified. CoAP, MQTT (MQTT-SN version) and DDS are able to run on top of UDP, while AMQP and XMPP rely on TCP.

There are not many performance evaluation publications comparing these protocols [AFGM$^+$15]. With high packet-error rate CoAP delivers messages with lower delay than MQTT, but with low error rate CoAP's delay is higher. In a smartphone environment CoAP outperforms MQTT in terms of bandwidth usage and round-trip time. CoAP is able to outperform HTTP in terms of transmission time and energy usage in a wireless sensor setting [AFGM$^+$15].

2.5 Congestion and congestion control

Congestion occurs in a packet switching network under heavy load, when packets are sent at a higher rate than can be delivered or buffered, increasing delay and packet drops. Congestion is a problem especially in complex networks that have links of differing bandwidth [Nag84]. Packet drops itself are not the main problem, as drops are used as an indication of congestion [RFB01]. If congestion is not addressed properly, the network experiences a congestion collapse [Nag84, Jac88].

Before discussing congestion collapse, a short description of reliable transmission.

To ensure data delivery, TCP (among other protocols) uses a retransmission timer, which value is called the retransmission timeout (RTO) [PACS11]. If the TCP sender does not receive an acknowledgement after a certain time, the packet is resent. The RTO may be more conservative than the algorithms defined in RFC 6298, but not more aggressive in order not to cause congestion.

Many reliable transport protocols, including TCP, perform round-trip time (RTT) estimation to calculate a proper RTO value [KP87]. Because the Internet consists of a variety of networks, the RTTs are widely different. Congestion also fills up buffers that take time to drain, increasing delay. The computed RTO should be an upper bound on the actual RTT. If the RTO timer is set to less than actual RTT, a spurious RTO and unnecessary retransmission occurs.

Backoff of the RTO means that when a timeout occurs, the previous RTO is increased by a certain factor and the packet is retransmitted. This allots time for the congestion to deplete before the next retransmission.

Congestion collapse is a stable condition, where the network transmits mostly unnecessary retransmissions [Nag84]. Under load the RTTs in the network increase and if the RTT increases above the retransmission interval of a host, it will unnecessarily retransmit copies of the packet. The copies cause more congestion, causing the retransmission timer to expire more than once if RTO is not properly adjusted. Eventually the network is filled with unnecessary retransmissions and the throughput is severely reduced. A congestion collapse happened in the Internet in 1986 [Jac88].

In addition to the RTO algorithm, there are other approaches to congestion control. TCP has congestion control mechanisms, such as slow start and fast retransmit [APB09]. For example, slow start is congestion control at the beginning of the connection, increasing transmission rate until congestion occurs.

UDP does not contain built-in congestion control, thus congestion control has to be implemented in applications or protocols such as CoAP. The RFC 5405 [EF08] provides congestion control guidelines for UDP applications. The default congestion control scheme in CoAP (DefaultCoAP) is based only on a timer and exponential backoff without RTT estimation.

Active Queue Management techniques include Random Early Detection (RED) [BCC$^+$98] that drops packets probabilistically when the average queue size increases, indicating the sender about incipient congestion early on. The algorithms are im-

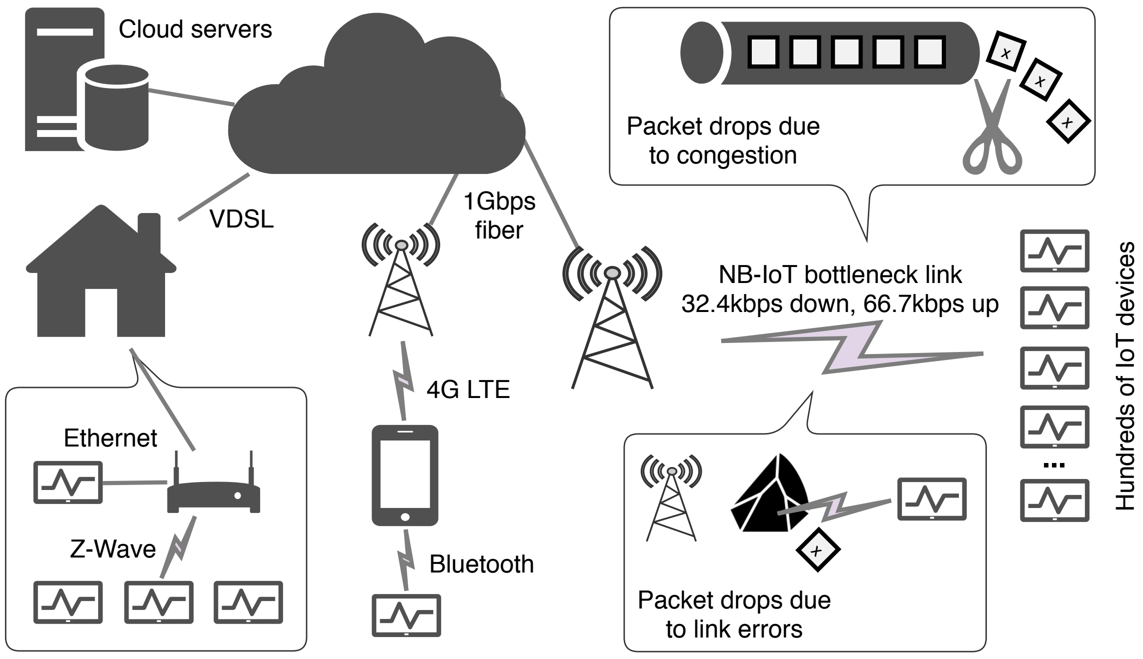
Cloud servers
VDSL
1Gbps fiber
Packet drops due to congestion
NB-IoT bottleneck link 32.4kbps down, 66.7kbps up
Hundreds of IoT devices
4G LTE
Ethernet
Z-Wave
Bluetooth
Packet drops due to link errors

Figure 3: Overview of an IoT network

plemented in routers, thus requiring support from the network administrators. Active queue management prevents single connections from monopolizing the queue (lock-out), lowers delay by keeping the buffers drained and reserves space for packet bursts.

Instead of dropping packets early on, the packets can be complemented with an indication of congestion using Explicit Congestion Notification (ECN) [RFB01]. An algorithm such as RED sets the Congestion Experienced codepoint in the IP packet header. In addition to routers, ECN requires support from the transport protocol (e.g. TCP) to reduce transmission rate, but ECN can be incrementally deployed. Avoiding packet drops is beneficial, for example, in latency-sensitive real-time audio and video streaming, where the retransmission delay would be excessively high.

Figure 3 represents a setup with mostly wireless IoT devices communicating with a server over the Internet. The left half of the figure depicts wireless and wired devices connected to the Internet via a gateway device. The gateway can provide security features and convert short range communication technologies (Z-Wave and Bluetooth) to long range wireless (4G LTE) or wired (VDSL) communications. The right half of the figure closely represents the test arrangement used in this thesis, where hundreds of devices share a long range bottleneck link, such as NB-IoT.

3 Baseline congestion control algorithms for CoAP

This section covers three existing congestion control algorithms that can be used with CoAP. The congestion control mechanism built into CoAP is called DefaultCoAP in this thesis. CoCoA [BBGD17] is an improved congestion control algorithm made specifically for CoAP. The congestion control of CoAP over TCP is provided by TCP, thus the TCP retransmission timer algorithm is discussed.

3.1 DefaultCoAP

The trade-off of simple algorithms is their possible low performance. For reliable message delivery, the CoAP protocol introduces a no frills congestion control algorithm. By default the algorithm initializes the retransmission timeout (RTO) timer to a random value between 2 to 3 seconds [SHB14] for each new message. If no acknowledgement is received, the message is retransmitted and the RTO is doubled on timer expiration. The timer is reset after an acknowledgement is received or retransmission attempts are depleted. The randomness (dithering) in the initial RTO is used to prevent synchronization effects.

This approach has its problems. RFC 3481 [IML^{+}03] suggests that initial RTO value should not be less than 3 seconds, because 2.5G/3G networks may have high latencies. In such a case, the original transmission inevitably encounters a spurious RTO. The RTT of a network may exceed 3 seconds especially in the presence of bufferbloat [GN11].

3.2 TCP retransmission timer algorithm

The basis of the reliable data transmission in TCP is the retransmission timer. If a message is not acknowledged, the message is retransmitted on timer expiration. The retransmission timeout (RTO) value is calculated with the algorithm specified in RFC 6298 [PACS11].

Round-trip time (RTT) is the time passed from the transmission to the receiving of the acknowledgement. RTT sampling uses Karn's algorithm [KP87] that does not include samples from retransmitted messages, because when a packet has been retransmitted, in the traditional scheme it is not known that which transmission is actually acknowledged. This is called retransmission ambiguity. If it is assumed that the acknowledgement is to the most recent retransmission, it is often a false assumption. Assuming the RTT estimate is correct, the original acknowledgement might arrive very shortly after the spurious retransmission, causing too small RTTs to be sampled repeatedly. If the sample is measured from the original transmission and the path is lossy, the RTT estimate becomes too large. If RTT samples are not measured from retransmitted packets, then in case of an increase in the actual RTT, the RTT estimate does not grow at all because all packets are spuriously retransmitted at least once. Karn's algorithm solves the problem by ignoring measurements from retransmitted packets, but keeping the backed off RTO value for the next new message to increase the probability of measuring an unambiguous sample.

In the RFC 6298 algorithm, the initial RTO is 1 second or more. The SRTT (smoothed RTT) variable contains a smoothed average of the RTT samples, where the latest sample has the largest weight. The RTTVAR (RTT variation) variable contains the smoothed average of RTT deviation. When the first RTT is sampled, the SRTT is initialized to the RTT sample and the RTTVAR to half of the RTT sample. The RTO value is then calculated as:

$$RTO = SRTT + max(G, K * RTTVAR) \tag{1}$$

Where G is the clock granularity in seconds and K is 4 by default.

On subsequent RTT samples the SRTT and RTTVAR variables are updated:

$$RTTVAR = (1 - beta) * RTTVAR + beta * |SRTT - sample| \tag{2}$$

$$SRTT = (1 - alpha) * SRTT + alpha * sample \tag{3}$$

Where alpha should be 1/8 and beta should be 1/4. The RTO is then calculated using the equation 1.

If the calculated RTO value is less than 1 second, it should be set to 1 second. A TCP sender can be more conservative. The SRTT and RTTVAR variables may be cleared if too many backoffs occur, indicating a change in the network. There is no dithering on the RTO value, causing possible synchronization issues [GAMC18].

In cases where only one message is in flight at a time, as in CoAP traffic, the RTO recovery mechanism is the only one available. Mechanisms such as Fast Retransmit [Ste97] cannot be used.

3.3 CoAP Simple Congestion Control/Advanced (CoCoA)

To combat the shortcomings of the default congestion control algorithm, an alternative congestion control algorithm called CoAP Simple Congestion Control/Advanced (CoCoA) has been proposed [BBGD17]. The CoAP specification allows the in-place replacement of the congestion control algorithm, making the switch to alternative algorithms straightforward.

CoCoA uses round-trip time (RTT) measurements to estimate the actual RTT of the link. The estimates are then used to compute more suitable RTO timer values to gain better performance.

CoAP is used in various network types that have different RTT values. Before any RTT samples have been measured, CoCoA uses 2 seconds initial RTO value. With dithering the range is the same as with DefaultCoAP, as dithering is from RTO to ACK_RANDOM_FACTOR * RTO (default 1.5).

The RTO calculation is based on the algorithm defined in RFC 6298 [PACS11] with some modifications. Two RFC 6298 estimators are run in parallel: the strong estimator is updated only when the exchange required no retransmissions, and the weak estimator is updated when no more than two retransmissions were required. As there is ambiguity in which of the transmissions was acknowledged, the update to the weak estimator calculates the time taken from the original transmission. The RTO used for the next new message is an (exponentially) weighted moving average of the two estimators using the following weights:

RTO := 0.5 * E_strong_ + 0.5 * RTO (used when the strong estimator is updated)

RTO := 0.25 * E_weak_ + 0.75 * RTO (used when the weak estimator is updated)

The initial values for the estimators are 2 seconds. The factor K in the weak estimator is set to 1 to avoid too steep increases when a sample is measured from retransmitted exchanges.

CoCoA uses a variable backoff factor to avoid using up all retransmissions too quickly or too slowly. RTO estimates shorter than 1 second are backed off with a factor of 3 to conserve retransmissions. For RTOs between 1 and 3 seconds the factor is 2 as in DefaultCoAP and TCP, and for RTOs larger than 3 seconds the factor is 1.5.

The exponential backoff is truncated at 32 seconds . The RTO estimator can have values larger than 32 seconds and that is not limited by the specification nor by the implementation used in the tests.

The CoCoA specification states that the RTO value must be decayed during idle periods to take into account possible changes in the network, such as a switch from LTE mode to GPRS mode in a cellular network. The estimate is decayed towards the initial RTO of 2-3 seconds. If the RTO value is lower than 1 second, and it has not been updated in 16 times the value, double the RTO. If the RTO value is higher than 3 seconds, and it has not been updated in 4 times the value, set the RTO as 1 second + RTO/2.

4 Related work

There are a number of studies that have analyzed congestion control algorithms and the behaviour of network protocols in IoT environments. This thesis concentrates on the CoAP protocol, which is the subject of many other studies. IoT devices have special requirements that require special test cases. Thus the new experiments are fundamental, but the previous results of others are discussed here.

The number of clients in the related papers are quite small. The CoAP maximum retransmissions parameter value is increased from the default 4 to 10 in [JDK15] with maximum 80 clients.

CoCoA has been developed as part of research work and there are multiple papers published by the authors [BBGD17]. Only the most recent ones are discussed, since there have been modifications to the protocol, such as setting 32 seconds maximum backed off RTO instead of earlier 60 seconds maximum, and that the weak estimator is updated when a maximum of two retransmissions had occurred.

The test setup of [BGDP16] has two scenarios and includes evaluation of also other algorithms in addition to DefaultCoAP, CoCoA and TCP/Linux RTO. The scenario using an IEEE 802.15.4 multihop network is quite different from the one used in this thesis. The second scenario consists of a computer using actual GPRS link connecting to a fixed server over the Internet, where this link is the only wireless one. The bandwidth of the link is approximately 40 kbps downlink and 15 kbps uplink. The clients and server use Java Californium (Cf) CoAP implementation. Their TCP tests cover only the congestion control algorithm (Linux RTO), as the tests are carried over UDP with the ordinary CoAP UDP header.

The workloads are a divided into continuous and burst traffic workloads. The continuous workload consists of 10 to 40 clients that exchange request-response pairs with the server back to back for 180 seconds. The burst traffic workload consists of 10 continuous clients and a sudden inclusion of up to 50 continuous clients.

The packet loss rate of GPRS is relatively low and the performance of the continuous workload depends mostly on adaptation to the high RTT. The RTT increases due to queuing delay as the amount of clients increases. DefaultCoAP's fixed RTO values cause spurious RTOs and react slowly to actual losses. CoCoA performs better than DefaultCoAP and Linux RTO. Linux RTO does not increase the RTO value when the RTT decreases, leading to too small RTO values when congestion is present. With 40 clients the average finished transactions per second (throughput, the higher the better) are about 16.5 for DefaultCoAP, 19.0 for CoCoA and 17.8 for Linux RTO.

Similarly with the burst traffic load, CoCoA performs the best while DefaultCoAP performs the worst. CoCoA's variable backoff and aging mechanisms prevent the RTO value from growing too large when sequential packets are lost due to link errors. With 30 burst clients, the average settling times in seconds (the lower the better) are about 180 (truncated) for DefaultCoAP, 105 for CoCoA and 110 for Linux RTO.

These results are in line with our results. With high prevailing RTT, DefaultCoAP does unnecessary retransmissions with each new message as it does not adapt to the high RTT. In our error-prone tests, even with the low error rate, DefaultCoAP recovers from packet losses slower than CoCoA and TCP. In our error-free scenario with 50 clients, the results with DefaultCoAP and CoCoA are roughly the same, though. While CoCoA's variable backoff and aging mechanisms help in the error-prone scenarios, they cause CoCoA clients to behave too aggressively wasting the scarce bandwidth in error-free scenarios. When facing bufferbloat, aging prevents the RTO value from increasing to a proper level by quickly decaying it back to the default value range.

The paper [JDK15] evaluates DefaultCoAP, CoCoA, Linux RTO (UDP as previously) in addition to Peak-Hopper. The test setup emulates a single hop ZigBee network using Netem with Java Californium as the CoAP implementation. The bandwidth of the bottleneck link is 20 kbps with a 512-byte buffer and it is error free. The workloads consist of 1 to 80 clients that communicate with a server over the shared link, exchanging 50 request-response pairs back to back with 30 byte response payload. Continuous clients exchange the 50 pairs normally and random clients reset their congestion control state after 1 to 10 exchanges. They also mixed the client types and DefaultCoAP clients with other congestion control clients for competition testing.

The median client completion time (similar to flow completion time) of DefaultCoAP clients is stable but usually the highest, because its RTO value is high for the link. Continuous CoCoA and Linux RTO clients use RTT measurements to lower the RTO value and to complete quicker. Random CoCoA clients reset the state often enough for the completion time to be only slightly lower than with DefaultCoAP. Linux RTO outperforms DefaultCoAP and CoCoA with 20 and more clients due to CoCoA's weak sampling increasing the RTO value unnecessarily high. Linux RTO is highly unstable and does have higher upper percentiles, though.

Our tests indicate similar unstableness and low median completion time of continuous Linux RTO (albeit using TCP) clients with small buffer sizes.

5 Test arrangements

The test environment used in [JRCK18a], [JRCK18b] and [JPR$^+$18] emulates a scenario, where multiple adjacent IoT devices connect to a fixed server. For example, informative displays at bus stops update their timetables by requesting an update and then receiving incremental updates to the timetable. The devices share a constrained bottleneck link to a router connected to the Internet.

5.1 Test phenomena

If the path from the client to the server has plenty of available bandwidth, low delay and is error free, then the congestion control algorithm has little impact on the performance (aside from protocol options and different header sizes). However, the wireless links commonly used in IoT are constrained and error prone.

1. Losses due to congestion The common congestion case is one where the buffer of the bottleneck router is not large enough to hold all of the incoming packets. The packets that do not fit into the buffer are discarded (tail-dropped) when the network/link becomes congested.

A congestion control algorithm should decrease the transmission rate so that the buffer size is not exceeded and that the link stays fully utilized. The buffer should not be fully depleted, because then the link becomes idle at times, lowering its utilization.

2. Bufferbloat

Buffers are essential for packet networks, but too large buffers cause excessive delays [GN11]. Without buffers the packets have no place to wait for transmission and the network has to be globally synchronized, which is expensive and inflexible. Active queue management techniques such as Random Early Detection (RED) are used to keep the buffers from growing too large for fluent operation. Because active queue management is not widely enabled in network devices, memory chips are cheap and packet loss is minimized, bufferbloat emerges. As the buffer fills up, the delay increases linearly, causing slow reaction to congestion. Commodity network devices may use a single buffer size for all links connected to it. The buffer is then sized for the fastest connection, causing the buffer to be too large for slower connections, such as wireless ones. Reports indicate that bufferbloat exists also at the core of the Internet [GN11].

A congestion control algorithm should wait long enough for the packet to traverse through the buffer (queuing delay), that is, a longer RTO value is better.

3. Losses due to link errors

The link loses packets due to the corruption of physical layer frames [IML+03]. In the test network the losses are emulated and reproducible.

A congestion control algorithm should retransmit the packet quickly, i.e. shorter RTO value is better. As the packet is lost, the recovery method is to retransmit it as quickly as possible.

As can be noticed, the appropriate actions for the phenomena are contradictory: longer RTO for congestion and bufferbloat, but shorter RTO for recovering from link errors. Networks can suffer from each of the phenomena, making the design of

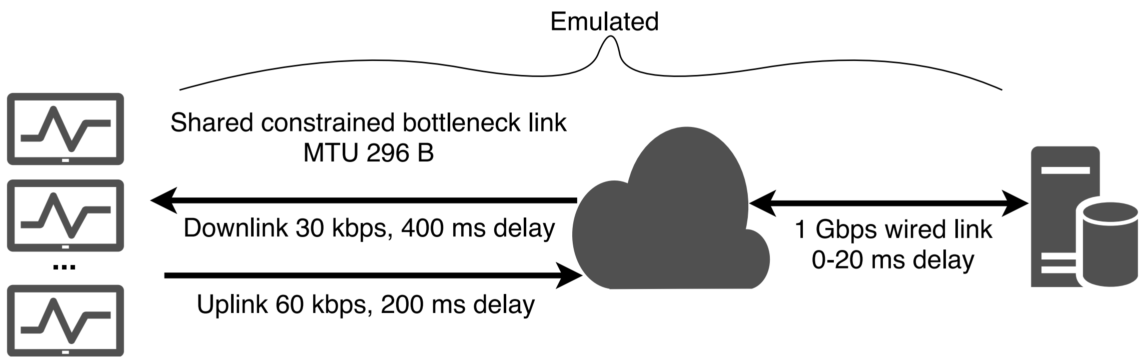
Emulated
Shared constrained bottleneck link
MTU 296 B
Downlink 30 kbps, 400 ms delay
Uplink 60 kbps, 200 ms delay
1 Gbps wired link
0-20 ms delay

Figure 4: The test setup

a congestion control algorithm for IoT environments challenging. The congestion control algorithm of TCP is designed for wired networks with mostly congestion related losses, and it performs sub-optimally under non-congestion losses [GAMC18]. Handling congestion is the main objective of congestion control and performance optimizations are secondary objectives [JRCK18a].

5.2 Test network

Network emulators create a constrained NB-IoT-like [CMHH17] link between the client and the server as shown in Figure 4. This bottleneck link has a data rate of 30 kbps downstream and 60 kbps upstream. There is a 400 ms delay downstream, 200 ms delay upstream and a random 10-20 ms delay between the bottleneck router and the server. The MTU of the bottleneck link is 296 bytes. The bottleneck router buffer size is selected from 2500 bytes, 14100 bytes, 28200 bytes and 1410000 bytes. The smallest one is approximately the bandwidth-delay product of the link and the largest one is practically infinite, i.e. large enough that packets are never lost due to congestion.

The test network consists of a client host running the client software, a server host running the server software and two hosts running the Netem network emulator. The clients and the server use actual implementations of the algorithms. The hosts are physical computers running Debian 9 Stretch. The client generates traffic that is typical for IoT devices. The server is a more powerful fixed host, such as a database server. The traffic is captured from different network interfaces using tcpdump for analysis.

Independent of the direction of the data flow, the first network emulator emulates the link bitrate and the buffer of the bottleneck router. The second network emulator then emulates propagation delay and packet loss. This arrangement ensures the correct emulation of buffers, link capacity, delays and packet drops.

The error free test scenarios are subject only to congestion-related losses, meaning that packet drops are only due to full buffers. With the infinite buffer size, the buffer of the bottleneck router is large enough to hold all sent packets. It takes considerable time to empty the buffer. All spuriously retransmitted packets are queued into the buffer, increasing the RTT and decreasing the throughput.

The link-error test scenarios have three different error profiles. The alternation and the average error rate is accomplished with a two-state Markov model that creates short error bursts of the higher error rate. The low error profile has a constant average 2% packet-error rate without state alternation. The medium error profile has an average 10% error rate using alternation between 0% (good state) and 50% (bad state). The high error profile has an average 18% error rate using alternation between 2% and 80%.

5.3 Workloads

The test requests contain the minimal CoAP header of 4 bytes with UDP or 2 bytes with TCP, and 6 bytes for the URI path option. The response CoAP header is 1 byte larger due to the payload marker. A zero length token is used for all messages, except for congestion control algorithms requiring a token value, where a 1-byte token is used. 8-byte UDP and 20-byte TCP transport headers are used in the tests in addition to a 20-byte IPv4 header.

The responses have a payload of 60 bytes. Such payload could contain for example temperature, electricity consumption or air quality measurements. Experiments with larger payloads, that are used for example with firmware updates, are not discussed in this thesis, but are in [Pes19].

The tests are conducted with two types of clients and the number of concurrent clients varies from 1 to 400. "Continuous" clients exchange 50 request-response pairs. "Random" clients exchange the 50 request-response pairs in batches of 1 to 10 messages and then reset all state information. This emulates a situation where the client sends only a few messages and is then followed by another similar client. Such short exchanges are challenging for congestion control, because the algorithms have to start with an initial RTO and have only little time to adapt to the prevailing RTT. Note that with DefaultCoAP the continuous and random clients function identically as DefaultCoAP does not have any state information.

In the CoAP over TCP tests with continuous clients, the three-way handshake is completed and the CSM message is sent beforehand and the timing is started at the transmission of the first request message. With random clients, this is not the case as the TCP connection is closed after each batch and a new TCP connection is established before sending the next batch.

5.4 Metrics

All of the error free tests are run for 20 replications and the error-prone tests are run for 40 replications. The metrics are calculated from these replications.

The main metric is the Flow Completion Time (FCT) which is the time taken for a single client to exchange the 50 request-response pairs. More specifically it is the time from the first request to the first response of the last request. It includes the TCP connection establishment for random clients.

Other metrics were also used while analysing the algorithms. RTT is the time taken to complete a single request-response pair. Frequency of transmissions is the number of transmissions required to complete a request-response pair. Expired RTO measures the RTO values and Client RTO max is the highest expired RTO. Number of packet drops is the difference between sent and received packets. Number of unnecessary retransmissions is the amount of late arriving response duplicates after the first (non-duplicate) response.

5.5 Modifications to default settings and implementations

Maximum number of retransmissions and RTO

The CoAP MAX_RETRANSMIT value, controlling the maximum number of retransmissions, is increased from the default 4 to 20 in order to complete the large tests without errors. The CoAP specification does not specify an upper bound limit for the backed off RTO value, so with maximum 20 retransmissions the value can get very high. The DefaultCoAP RTO value is thus limited to 60 seconds.

Linux TCP for CoAP over TCP

The used TCP implementation is the full-fledged Linux TCP implementation and not one for constrained devices, but it is configured as a more standardized version of TCP that is more suitable for constrained devices. The implementation uses TCP NewReno [HFGN12]. Control Block Interdependence (CBI) [Tou97], TCP Timestamp option [BBJS14] and TCP Fast Open (TFO) [CCRJ14] are not used. The TCP SYN and SYN/ACK retries is increased to 40, and delayed ACKs are sent with a constant 200 ms timer. The Linux TCP has by default an initial RTO of 2 seconds and a maximum RTO of 120 seconds, which were not modified.

Congestion control algorithms

The used libcoap version is 4.1.2. It includes only the DefaultCoAP algorithm. CoCoA, CoAP over TCP and the improved algorithms were implemented into libcoap. The implemented CoCoA version is draft-ietf-core-cocoa-03 [BBGD17]. CoAP has two layers, the messaging layer for reliable communication and request/response layer for REST communication. The new congestion control algorithms were implemented into the messaging layer. The aptly named functions that were affected are coap_send_confirmed (send new message), handle_response (receive acknowledgements and piggybacked responses) and coap_retransmit (retransmit message on RTO timer expiration).

TCP support to libcoap

CoAP over TCP functionality required the implementation of the CoAP over TCP header structure. Separate send and receive functions were implemented, but retransmissions and congestion control were not, because TCP handles them.

CoCoA aging

The CoCoA specification states that the RTO estimate must be decayed during idle periods. The concept of an idle period is only briefly mentioned. The idle period is thus interpreted as a state where there are no ongoing transmissions. There are no idle periods in our workloads as the next request is sent immediately after receiving a response. The aging mechanism is explicitly turned off in the used CoCoA implementation, since especially with the increased MAX_RETRANSMIT value, the test cases can take so long that aging would be performed on a busy flow.

6 Baseline results

DefaultCoAP, CoCoA and CoAP over TCP are empirically analyzed in [JRCK18a, JPR$^+$18] to find their shortcomings and to form a comparison baseline for the improved algorithms. The results with an error free link and an error-prone link are discussed separately. The results include both continuous and random workloads.

The figures in the results chapters contain flow completion times as boxplots. The whiskers are the 10th and 90th percentiles, the edges of the box are the 25th and

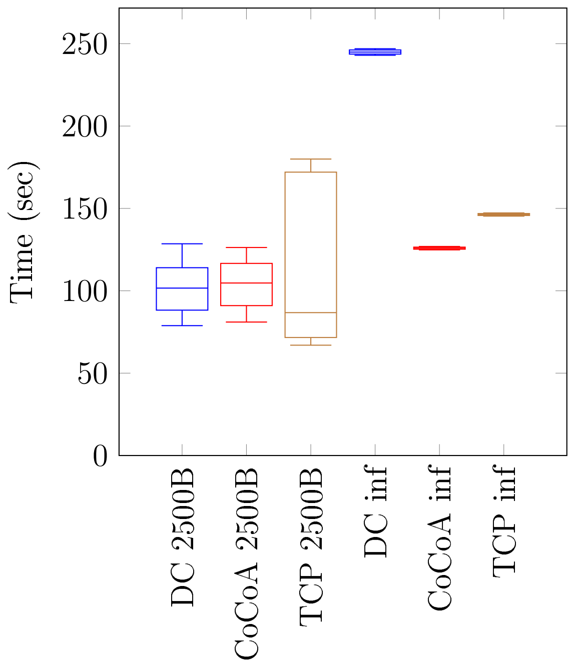

Figure 5: FCT of 100 continuous clients, error free link, using 2500B and infinite buffer sizes

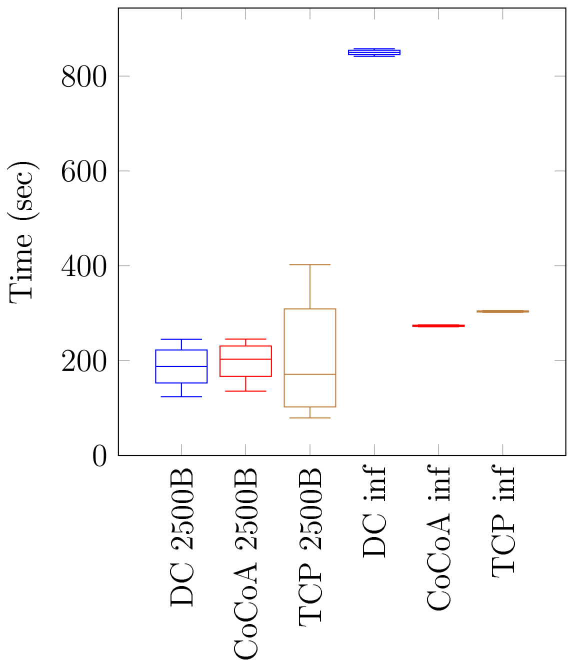

Figure 6: FCT of 200 continuous clients, error free link, using 2500B and infinite buffer sizes

75th percentiles (quartiles) and the line inside the box indicates the median. The algorithm and used buffer size is indicated on the x-axis. The names of the algorithms have been abbreviated: DefaultCoAP is DC and CoAP over TCP is TCP.

6.1 Error free link

Continuous workload

Starting with 100 continuous clients, the congestion with small buffer sizes results in packet losses. Using the small 2500B buffer size, the median FCT of DefaultCoAP is 101.6 seconds, CoCoA 104.7 seconds and CoAP over TCP 86.7 seconds (see Figure 5). CoAP over TCP reacts to congestion more effectively than DefaultCoAP and CoCoA. TCP keeps the backed off timer value until data is acknowledged without retransmissions (Karn's algorithm) decreasing congestion and lowering the amount of retransmissions. Both DefaultCoAP and CoCoA transmit the next message with non-backed off RTO value despite that retransmissions occurred and congestion may be present. This causes more packet drops and retransmissions. Only some CoAP over TCP clients have to back off, making way for other clients, and may have a very high FCT. Also, the TCP timer limit is 120 seconds instead of 60 as in the others, causing roughly 5% of the clients to use an RTO longer than 60 seconds. Even though it is unfair for the clients that have to wait longer, it is effective as the median FCT of CoAP over TCP is 14.7% and 17.2% shorter than DefaultCoAP's and CoCoA's, respectively.

When using the infinite buffer, the median FCT of DefaultCoAP is 244.911 seconds, CoCoA 125.870 seconds and CoAP over TCP 146.308 seconds. TCP's larger header causes enough overhead so that the CoCoA clients are able to complete faster. The queue is so long that the RTT becomes much higher than two seconds. As the initial RTO is at most 3 seconds for each algorithm, many of the first transmissions encounter a spurious RTO (TCP has already measured an RTT estimate during the connection establishment). For DefaultCoAP the situation is graver, because every transmission begins with a 2-3 seconds RTO, causing every new message to be unnecessarily retransmitted. CoCoA and CoAP over TCP have to unnecessarily retransmit only a couple of messages before the RTO value is set high enough.

With 200 clients the packet losses with small buffer sizes and queuing delay with large buffer sizes increases. With the 2500B buffer size, the median FCT of DefaultCoAP is 187.758 seconds, CoCoA 202.999 seconds and CoAP over TCP 171.142 seconds (see Figure 6). When packets are lost due to congestion, the median FCT of CoAP over TCP is the shortest because of TCP's responsiveness to congestion. CoCoA and TCP measure the RTT and adjust the RTO values accordingly, requiring only a few unnecessary retransmissions in the beginning.

With the infinite buffer size, the median FCT of DefaultCoAP is 849.945 seconds, CoCoA 273.486 seconds and CoAP over TCP 303.799 seconds. DefaultCoAP reacts to congestion by backing off the RTO value exponentially, which works with small buffers, but because the prevailing RTT with large buffer sizes is longer than the initial RTO, a lot of spurious RTOs occur. DefaultCoAP restores the initial RTO value after each successful exchange and fills the buffer with spurious RTOs increasing queuing delay even more. Because only little forward progress is made due to the transmission of unnecessary retransmissions, with the infinite buffer size the situation is similar to a congestion collapse.

When packets are not lost, TCP's larger header causes the median FCT to be longer than CoCoA's [JPR+18].

With 400 clients and the 2500B buffer size, the median FCT of CoAP over TCP is the shortest. The median FCT of DefaultCoAP is 386.756 seconds, CoCoA 411.899

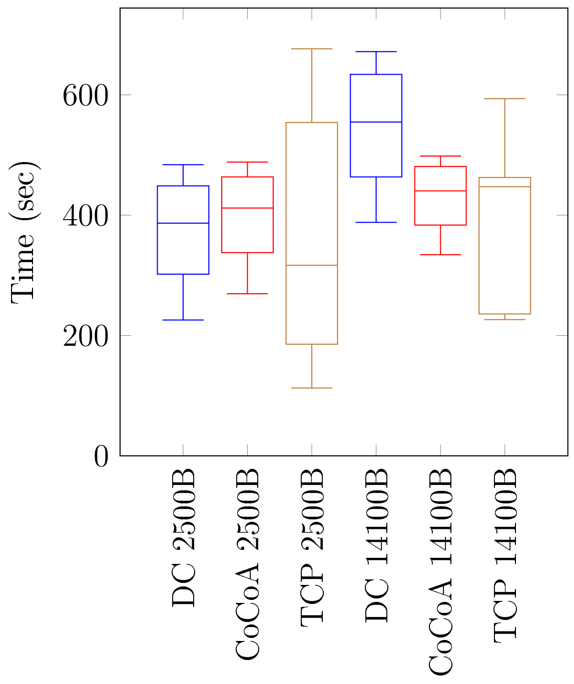

Figure 7: FCT of 400 continuous clients, error free link, using 2500B and 14100B buffer sizes

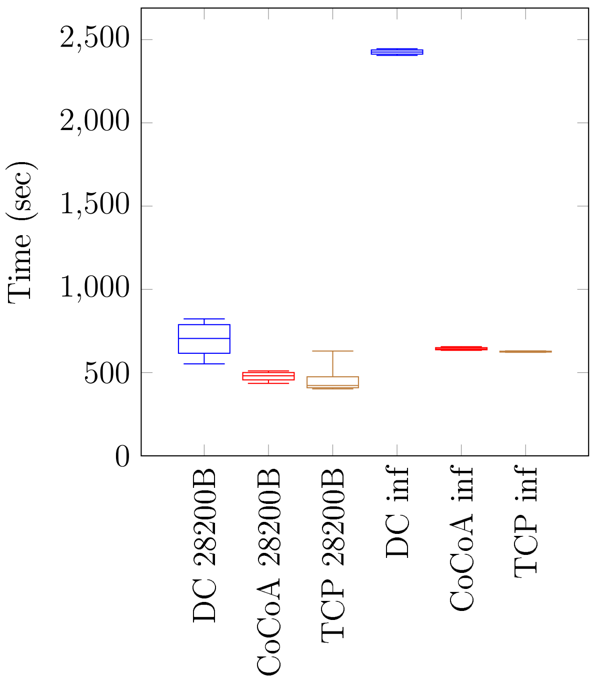

Figure 8: FCT of 400 continuous clients, error free link, using 28200B and infinite buffer sizes

seconds and CoAP over TCP 316.687 seconds (see Figure 7). The longer median FCT of CoCoA is due to CoCoA's aggressiveness: the variable backoff factor is only 1.5 when the RTO value is above 3 seconds, as it often is in this case. This causes the long RTO values to be backed off less than necessary and the message is retransmitted too quickly.

With the 14100B buffer size, the median FCT of DefaultCoAP is 554.857 seconds, CoCoA 440.427 seconds and CoAP over TCP 447.289 seconds. With the 28200B buffer size, the median FCT of DefaultCoAP is 705.121 seconds, CoCoA 480.779 seconds and CoAP over TCP 422.398 seconds. As the buffer size increases, more packets introduce more queuing delay.

With the infinite buffer size all transmissions are queued and every unnecessary retransmission wastes the bottleneck link. DefaultCoAP does on median 196 unnecessary retransmissions, wasting about 80% of the link capacity, and resulting in a congestion collapse. The median FCT of DefaultCoAP is 2425.320 seconds, CoCoA 642.100 seconds and CoAP over TCP 626.073 seconds (see Figure 8).

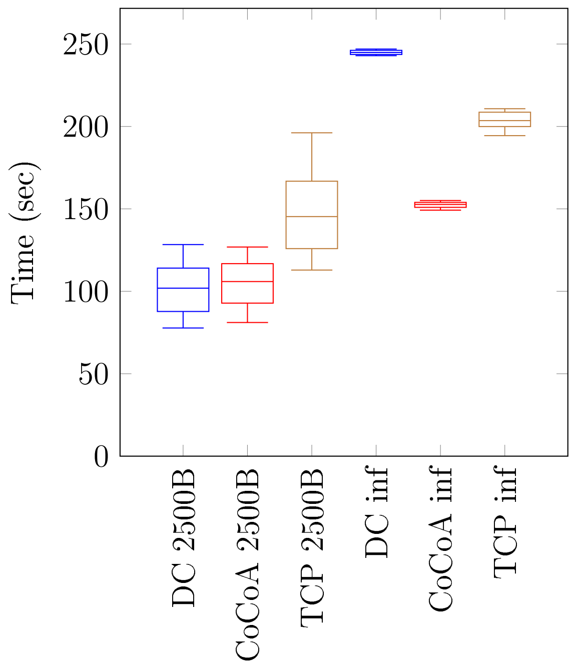

Figure 9: FCT of 100 random clients, error free link, using 2500B and infinite buffer sizes

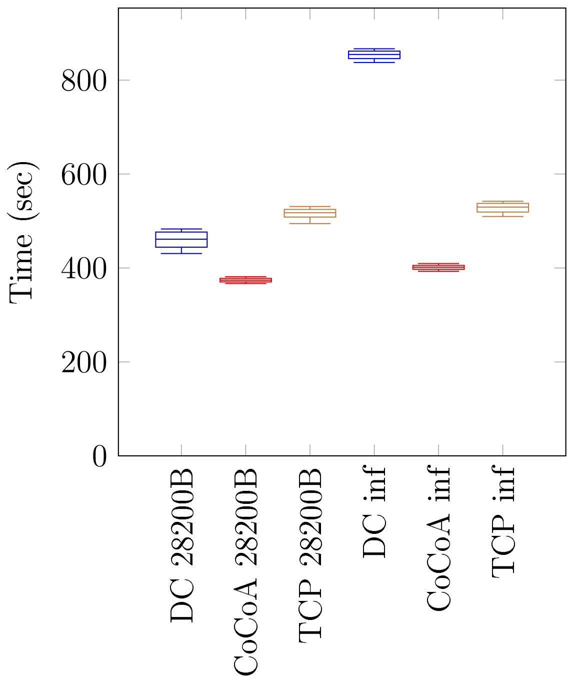

Figure 10: FCT of 200 random clients, error free link, using 28200B and infinite buffer sizes

Random workload

With the random workload the CoAP over TCP clients have to establish a new connection and exchange the Capabilities and Settings Message (CSM) at the beginning of each batch, which requires altogether two RTTs due to Nagle's algorithm [Nag84]. TCP's three-way handshake is a known shortcoming for typical IoT communication [JPR+18]. This causes the median FCT of random CoAP over TCP clients to be 47.093 seconds with one client, that is 41.8% higher than the median FCT of continuous CoAP over TCP clients [JPR+18].

With 100 random clients and the small 2500B buffer, the results of DefaultCoAP and CoCoA are similar to the results of the continuous clients (see Figure 9). The median FCT of DefaultCoAP is 101.908 seconds, CoCoA 105.918 seconds and CoAP over TCP 145.320 seconds. When using the small buffer size, TCP's connection establishment packets can also be dropped due to congestion and when using the infinite buffer, these packets cause more queuing delay.

When using the infinite buffer size, where the RTT is above two seconds, CoCoA and CoAP over TCP have longer median FCTs than continuous clients because

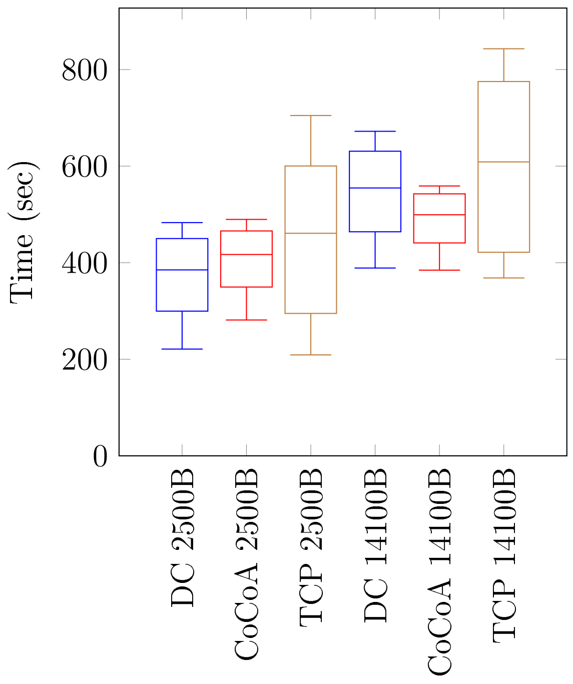

Figure 11: FCT of 400 random clients, error free link, using 2500B and 14100B buffer sizes

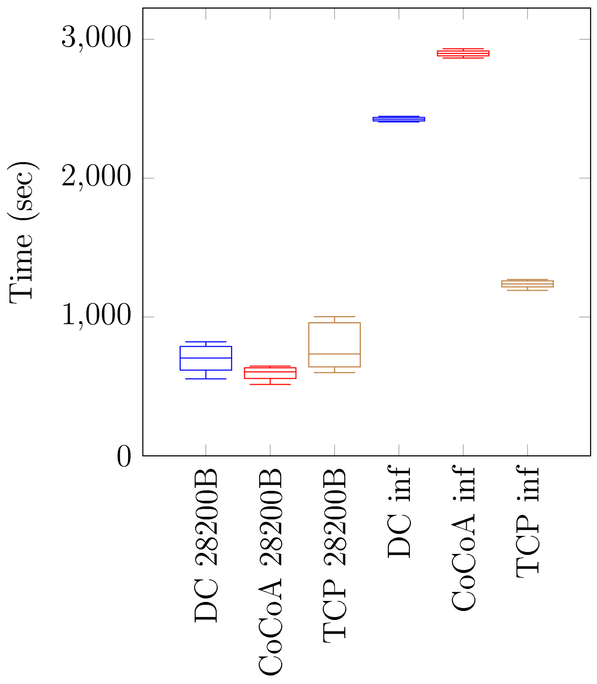

Figure 12: FCT of 400 random clients, error free link, using 28200B and infinite buffer sizes

their state is reset after each batch. The initial RTO value is too low causing spurious RTOs. The median FCT of DefaultCoAP is 244.863 seconds, CoCoA 152.693 seconds and CoAP over TCP 203.548 seconds.

With 200 random clients and the larger buffer sizes CoCoA slows down because the weak samples do not adjust the RTO value quickly enough, causing the RTO value to be too low and messages are spuriously retransmitted, increasing the queue (see Figure 10). This happens with continuous clients too, but only once per client. With the random workload every new batch starts with too low RTO value. The median FCT of CoAP over TCP is longer due to the additional three-way handshakes.

With the 28200B buffer size, the median FCT of DefaultCoAP is 461.338 seconds, CoCoA 374.131 seconds and CoAP over TCP 517.621 seconds.

With 400 random clients the trend is similar as with 200 random clients. With the 2500 buffer size, the median FCT of DefaultCoAP is 385.086 seconds, CoCoA 417.007 seconds and CoAP over TCP 460.947 seconds (see Figure 11). CoAP over TCP is the slowest one to complete.

With the 28200B buffer size the queue is long enough that the initial RTO is too small (see Figure 12). All algorithms do spurious RTOs. The median number of unnecessary retransmissions per client (exchanging only a few messages) is 64 for DefaultCoAP, 25 for CoCoA and 6 for CoAP over TCP.

With the infinite buffer size, the median FCT of DefaultCoAP is 2424.610 seconds, CoCoA 2898.480 seconds and CoAP over TCP 1237.885 seconds. Both DefaultCoAP and CoCoA experience a congestion collapse. Due to the long buffer queue, later CoCoA batches cannot get even weak RTT samples. CoCoA becomes an aggressive DefaultCoAP variant (the variable backoff factor is 1.5 instead of DefaultCoAP's 2). CoCoA unnecessarily retransmits each message 5 to 6 times.

CoAP over TCP does a couple of unnecessary retransmissions when establishing the connection and when transmitting the CSM message, but the total number of unnecessary retransmissions is much lower than DefaultCoAP's and CoCoA's [JPR+18].

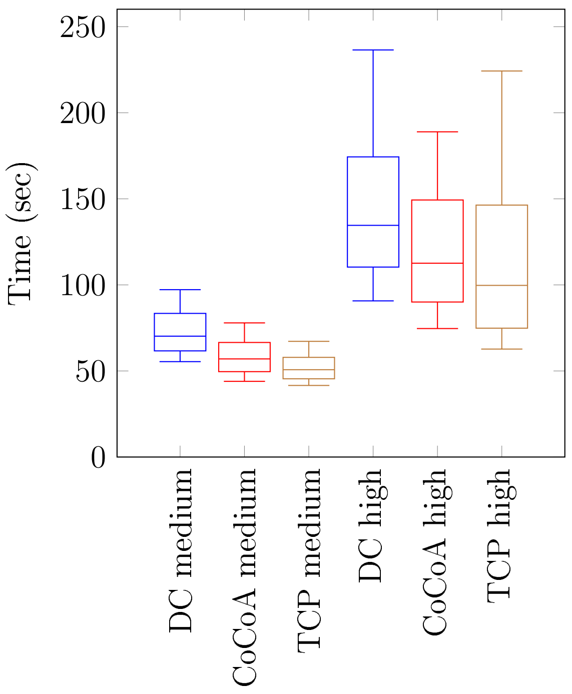

Figure 13: FCT of 10 continuous clients, error-prone link, using medium and high error rates

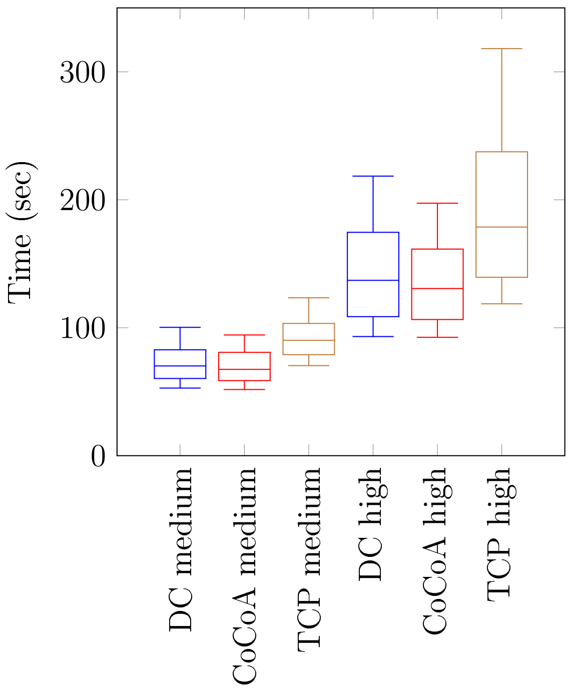

Figure 14: FCT of 10 random clients, error-prone link, using medium and high error rates

6.2 Error-prone link

In the error-prone test cases, packet losses are due to link errors and not congestion. For baseline, using the continuous workload without errors and with 10 clients, the median FCT of DefaultCoAP is 33.220 seconds, CoCoA 33.220 seconds and CoAP over TCP 33.440 seconds. With low error rate the median FCT of DefaultCoAP is 38.126 seconds, CoCoA 35.531 seconds and CoAP over TCP 35.341 seconds.

When the error rate increases, the FCT grows. With medium error rate the median FCT of DefaultCoAP is 70.212 seconds, CoCoA 57.002 seconds and CoAP over TCP 50.726 seconds (see Figure 13).

The median FCT of CoAP over TCP is the shortest one with medium error rate. TCP's RTT estimate is more accurate than CoCoA's (and DefaultCoAP does not perform RTT measurements), which allows quicker retransmission of lost packets. CoCoA's strong RTT estimate is weighed with only 0.5 causing slow convergence and the weak estimator inflates the RTO value. Both of these cause the RTO value to be too high and the reaction to lost packets to be slow [JPR+18].

With high error rate the median FCT of DefaultCoAP is 134.596 seconds, CoCoA 112.606 seconds and CoAP over TCP 99.721 seconds.

With high error rate the upper percentiles of CoAP over TCP's FCT are the high due to Karn's algorithm [KP87]: if consecutive messages are dropped, the RTO is unnecessarily backed off. Due to the lack of Karn's algorithim in DefaultCoAP and CoCoA, their poor performance with congestion allows them to recover losses more quickly [JPR+18].

Using the random workload without errors and with 10 clients the median FCT of DefaultCoAP is 33.220 seconds, CoCoA 33.220 seconds and CoAP over TCP 47.598 seconds. With low error rate the median FCT of DefaultCoAP is 38.260 seconds, CoCoA 37.193 seconds and CoAP over TCP 52.015 seconds.

CoAP over TCP's median FCT is the longest due to the TCP connection establishment segments as with the error-free clients. These segments are also subject to packet losses [JPR+18].

With medium error rate the median FCT of DefaultCoAP is 70.148 seconds, CoCoA 67.484 seconds and CoAP over TCP 90.179 seconds (see Figure 14). With high error rate the median FCT of DefaultCoAP is 137.065 seconds, CoCoA 130.645 seconds and CoAP over TCP 178.696 seconds.

The error-prone link with the random workload is the most realistic and demanding one. Improvements in the FCT of these cases are very desirable.

6.3 Summary

DefaultCoAP does not use RTT measurements to adjust the RTO and thus does unnecessary retransmissions when the prevailing RTT is larger than the initial RTO of 2 to 3 seconds. This is the case especially with 400 clients and infinite buffer size.

CoCoA's problems include adjusting the RTO estimate slowly and failing to respond properly to congestion by using a too low RTO value after an unnecessarily retransmitted message [JRCK18a]. With 400 random clients and an infinite buffer size, CoCoA clients fail to measure even weak samples, causing CoCoA to behave even more aggressively than DefaultCoAP.

The larger header and the three-way handshake of TCP hampers performance, especially in the random workload. Karn's algorithm makes TCP congestion safe, but it will slow down error recovery in error-prone cases because link errors are interpreted as congestion, making it less ideal for IoT communication.

7 Fast-Slow Retransmission Timeout (FASOR)

After analysing the shortcomings of the baseline algorithms, a new congestion control algorithm for IoT communications was created [JRCK18b]. The development was driven by empirical research. During development, the algorithm variants were empirically tested in the same network and with the same workloads as the baseline algorithms.

The objective was to create an algorithm that is congestion safe in the presence of congestion and bufferbloat, and also performs well in environments with link errors. The resulting algorithm is called Fast-Slow Retransmission Timeout (FASOR) and it can be complemented with accurate RTT measurements [JKRC18]. FASOR is based only on RTO measurements and logic, where FASOR with accurate RTT measurements requires additional information to be inserted into the message.

7.1 Development path

The development path consists of three variants from different development phases, each adding extra logic to the previous one. The variants are "Slow", "FastSlow" and "FastSlowFast". "FastSlowFast" is used by FASOR.

The improved algorithm is based on UDP because it has a lower header overhead and avoids the three-way handshake, which hampers the random workload. As with DefaultCoAP and CoCoA the losses are inferred using an RTO timer.

The algorithm uses the TCP RTO calculation formula (RFC 6298), requiring two state variables: SRTT and RTTVAR. The RTO value is initialized to 2000 ms. On each retransmission the RTO is doubled (grows exponentially). The RTO values are dithered between RTO + SRTT/4 and RTO + SRTT. The initial RTO is dithered with SRTT being set to 2/3 seconds. Similarly to DefaultCoAP and CoCoA, when the RTO value is backed off, it is not dithered again as the value is already dithered.

7.1.1 Improved backoff logic

A third state variable, SLOW_RTO, is used to store the time measurement from the original transmission to the first acknowledgement if retransmissions occurred. This value serves the same purpose as the Karn's algorithm: the backed-off RTO value is retained between transmissions in order to increase the probability of getting an accurate RTT measurement. This reduces the amount of unnecessary retransmissions in bufferbloat cases, but slows down error recovery in link error cases. The SLOW_RTO value is multiplied with a factor of 1.5 by default. The maximum RTO value is limited to 60 seconds. The SLOW_RTO value is dithered when used.

When an acknowledgement arrives, depending on the retransmission count; either the RTO value is updated per TCP RTO calculation algorithm (no retransmissions), or the total elapsed time value multiplied with 1.5 is stored to a SLOW_RTO variable (there were retransmissions). If there were no retransmissions, the next new message will have an RTO value calculated using the TCP RTO algorithm, but if there were retransmissions, the RTO value will be set to the value in the SLOW_RTO variable.

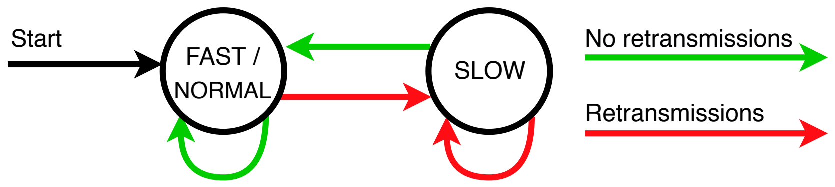

Figure 15: States and transitions of the "Slow" variant

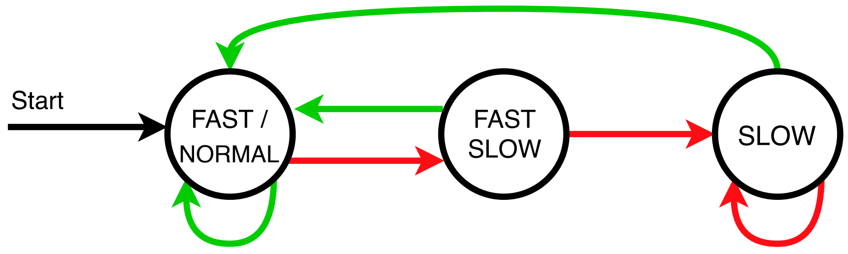

Figure 16: States and transitions of the "FastSlow" variant

The "Slow" variant consists of the TCP RTO calculation and a state where the SLOW_RTO variable is used (see Figure 15). The sender changes the state after a message has been acknowledged. If the transmission required no retransmissions, the green path is taken, and with retransmissions, the red path is taken. The "FAST/NORMAL" state arms the RTO timer with and backs off the TCP RTO. The "SLOW" state uses the SLOW_RTO value instead of the TCP RTO value.

To combat link errors more efficiently, a link error probe logic is introduced in the "FastSlow" variant (see Figure 16). The next new message will have a small RTO value in case the previous message was lost due to link errors and not congestion. More specifically, if the previous message had to be retransmitted, the RTO of the following message will be calculated using the TCP RTO algorithm instead of using the SLOW_RTO variable. The first retransmission of that message will have an RTO based on the SLOW_RTO value. In addition, the next new message will get the SLOW_RTO value as its RTO if the previous message had to be retransmitted.

"FastSlowFast" is the final variant and it is used by FASOR. "FastSlowFast" is similar to "FastSlow", but instead of backing-off the SLOW_RTO value, the TCP RTO value is backed-off. The RTO value sequence is TCP RTO, SLOW_RTO, TCP RTO * 2^1, TCP RTO * 2^2 and so on. It uses the SLOW_RTO value only to ensure that the sender does not cause a congestion collapse, since while waiting for the SLOW_RTO all the previous retransmissions have time to progress through the network with bufferbloat. This speeds up error recovery in the link error cases even more in case consecutive transmissions are lost.

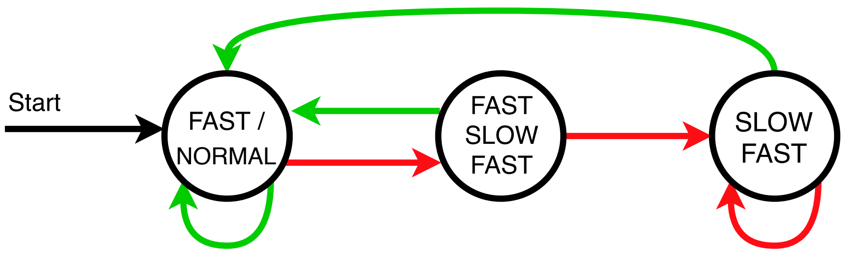

Figure 17: The FASOR states and transitions, adapted from [JRCK18b]

Figure 17 depicts the FASOR state diagram. The "FAST/NORMAL" state arms the RTO timer with and backs off the TCP RTO. The "FAST SLOW FAST" state starts with the TCP RTO, then uses the SLOW_RTO and backs off the TCP RTO. The "SLOW FAST" state starts with the SLOW_RTO and backs off the TCP RTO.

7.1.2 Fine tuning of the RTO calculation

Two small modifications to the TCP RTO calculation formula are made to speed up error recovery in link error cases.

First, the RTO value can be smaller than 1 second, even though the RFC 6298 suggests that RTO values less than 1 second should be rounded up to 1 second. This is because the RTT of a link can be less than a second causing the error-prone cases to notice the loss overly late. For TCP this is not an issue because RTO recovery is only one of the recovery methods, but for CoAP it is the only one. As with TCP, but unlike CoCoA, the estimators are updated only with unambiguous samples.

Second, the initialization phase sets the RTTVAR to an unnecessarily large value. Because the CoAP exchanges are short and far apart, the convergence to realistic RTT estimation is slow [JRCK18b]. With short exchanges (random clients workload) the initial RTO values play an important role [JDK15].

When the first sample is measured, normally the RTTVAR is set to measurement/2, but now the RTTVAR is set to measurement/2K, where K is 4. The RTO after this first measurement will be measurement + measurement/2, effectively the same as setting the K as 1 in the initialization phase.

7.2 FASOR pseudocode

The "FastSlowFast" algorithm with fine tuning is called Fast-Slow RTO (FASOR). The states and transitions of Figure 17 are described in a pseudocode form in Listing 1. The "fastrto" variable depicts the RTO value calculated using the TCP RTO formula.

Listing 1: FASOR pseudocode, adapted from [JKRC18]

```
SLOWRTO_FACTOR = 1.5
var state = NORMAL_RTO
var fastrto = 2000ms + dither()
var slowrto
var original_sendtime
var retransmit_count
var backoff_series_timer

// Sending Original Copy and Retransmitting 'req'
send_request(req):
        original_sendtime = time.now
        retransmit_count = 0
        arm_rto(calculate_rto())
        send(req)

rto_for(req):
        retransmit_count += 1
        arm_rto(calculate_rto())
        send(req)

// ACK Processings
ack():
        sample = time.now - original_sendtime
        if (retransmit_count == 0)
                unambiguous_ack(sample)
        else
                ambiguous_ack(sample)
```

```
unambiguous_ack(sample):
        k = 4
        if (is_first_sample()) //initial K = 1
                k = 1
        fastrto = rfc6298_update(k, sample)
        state = NORMAL_RTO

ambiguous_ack(sample):
        slowrto = sample * SLOWRTO_FACTOR + dither()
        state = ambiguous_nextstate(state)

ambiguous_nextstate(state):
        switch(state):
                case NORMAL_RTO: return FAST_SLOW_FAST_RTO
                case FAST_SLOW_FAST_RTO: return SLOW_FAST_RTO
                case SLOW_FAST_RTO: return SLOW_FAST_RTO

// RTO Calculations
calculate_rto():
        return <state>_rtoseries()

// fast, fast*2^1, fast*2^2...
normal_rtoseries():
        switch (retransmit_count):
                case 0: return fastrto_series_init()
                default: return fastrto_series_backoff()

// fast, max(slow, fast*2), fast*2^1, fast*2^2 ...
fastslowfast_rtoseries():
        switch (retransmit_count):
                case 0: return fastrto_series_init()
                case 1: return MAX(slowrto, 2*fastrto)
                default: return fastrto_series_backoff()
```

```
// slow, fast, fast*2^1, fast*2^2 ...
slowfast_rtoseries():
        switch (retransmit_count):
                case 0: return slowrto
                case 1: return fastrto_series_init()
                default: return fastrto_series_backoff()

fastrto_series_init():
        backoff_series_timer = fastrto + dither()
        return backoff_series_timer

fastrto_series_backoff():
        backoff_series_timer *= 2
        return backoff_series_timer
```

7.3 FASOR with accurate RTT measurements

All of the previously discussed algorithms have divided the arriving acknowledgements into either unambiguous or ambiguous samples based on whether any retransmissions of the acknowledged message were made. Accurate RTT measurements can be obtained using a modified token value or a CoAP option, supplementing the FASOR RTO computation logic by making all samples unambiguous.

The token and option insert a retransmission number (ordinal number of the transmission) into a message. The sender stores the timestamps of the original transmission and of each retransmission. When the first arriving response to the outstanding request arrives, the echoed token or option value is inspected and the RTT estimate value is updated using the corresponding transmission timestamp. This results in very accurate RTT estimates with only a small overhead. The token and option should not be used together because that would be redundant.

The effect is achieved in the QUIC protocol with monotonically increasing packet numbers to resolve retransmission ambiguity [IS18]. The packet number is different for the original transmission and all retransmissions, thus the acknowledgement can be associated with the correct transmission easily. This approach does not introduce any transmission overhead. CoAP uses the message ID for duplicate detection and it must be the same in retransmissions in case an earlier acknowledgement is lost.

A token in CoAP is a client-local request identifier [SHB14]. The client generates the token for each request and the server must include the same token in the response if a token was present in the request. When communicating with multiple endpoints, the token is used to match the responses to requests. As the token is client generated, a retransmission number can be encoded in the token. It cannot be used for empty acknowledgement messages, but common piggybacked responses only.

Our test setup uses one byte tokens for each message. The original transmission has a token value of zero, the first retransmission has a value of one and so on. The endpoint echoes the unmodified token in the responses as per the CoAP specification, thus no modifications to the endpoint software are required.

The token field can contain a traditional token value, too, when the field is partitioned into a retransmission counter and a (slightly smaller) token value. If the traditional token value is not used in the application, the token can be omitted from the original transmission to reduce the overhead. The absence of the token in a response is then interpreted as the original transmission.

The retransmission number can alternatively be encoded in a CoAP option like TCP Timestamp Option for Round-Trip Time Measurement [BBJS14]. CoAP options are conceptually similar to TCP options and are optionally included after the base header [SHB14]. CoAP options can be either critical or elective. A critical option must be understood/supported by the endpoint or an error message is returned. Elective options can be ignored if not supported. The FASOR option is an elective one, since processing it is only beneficial for the RTT estimation, but not crucial. The options are encoded in Type-Length-Value format.

The use of the option requires more space than the token: the option number takes 0-2 bytes, the header 1 byte and the payload 1 byte. The original transmission does not need the option as it can have an implicit value of zero.

The option has to be supported in both endpoints. For confirming the support, the very first request carries a special option value with all payload bits set to 1 (also limiting the maximum retransmissions to 254). When the sender receives an acknowledgement containing the value 254 or any other explicit value, the option support is considered confirmed. From this point on messages missing the option are inferred as original transmissions. The algorithm acts as the regular FASOR if the option support is not confirmed.

8 Results with improved algorithms

The results are discussed in the same order as in the baseline results, this time focusing on the improvements made. The results are analyzed in [JRCK18a, JPR+18, JRCK18b]. The comparison includes DefaultCoAP, CoCoA, CoAP over TCP, FASOR and FASOR with accurate RTT measurements using a Token (FASOR+Token, or FASOR+T in the figures). Results with small amount of clients are also presented.

The Token and Option are similar in functionality, providing means for accurate RTT measurements. Token was selected for the discussion because of its smaller overhead of only one byte and because it does not require any special support from the receiver.

8.1 FASOR development path

The algorithms in these tests can produce RTO values less than one second, but the K variable (the RTT variance multiplier) is initially set to the default 4 in "Slow", "FastSlow" and FASOR initk4. FASOR and FASOR+Token use K = 1.

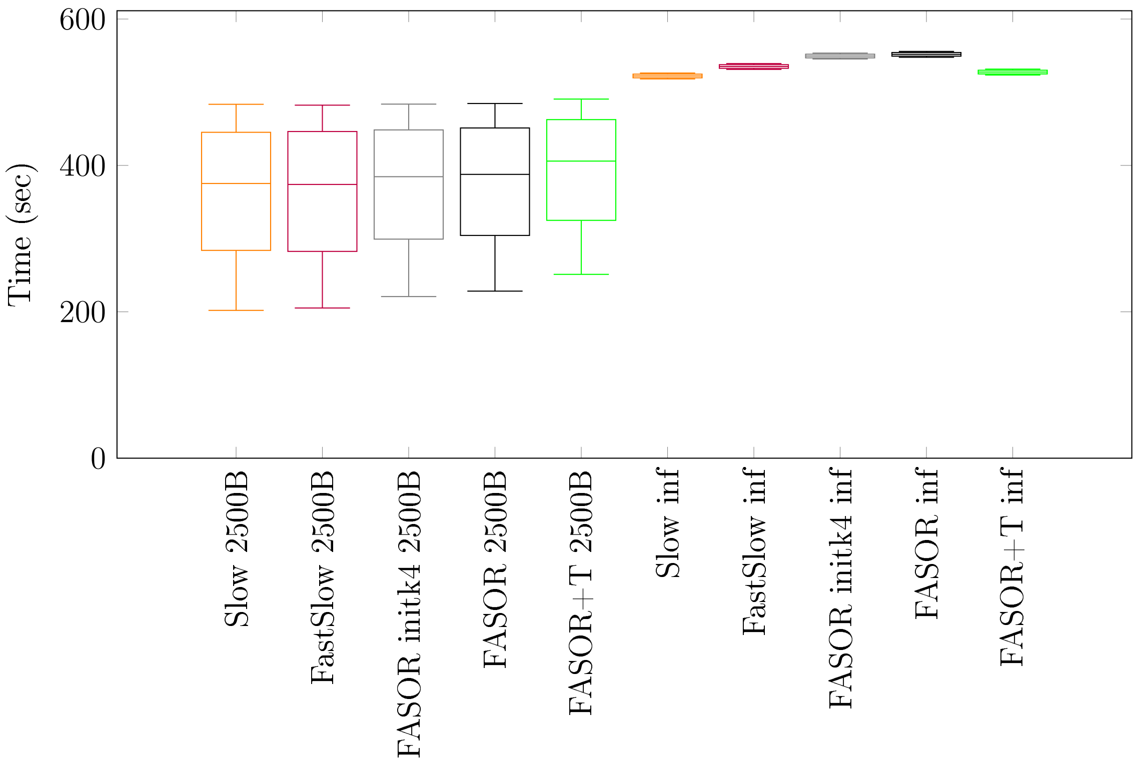

Figure 18: FCT of 400 continuous clients, error free link, development algorithms

"Slow" is the starting point providing a basic TCP RTO algorithm with RTT sampling, exponential backoff logic and Karn's algorithm using the SLOW_RTO variable. "Slow" provides good performance in environments where packet losses occur only due to congestion (see Figure 18).

"FastSlow" adds a probe for link errors by skipping the use of SLOW_RTO on the first new message after retransmissions. FASOR or "FastSlowFast" exponentially backs off the TCP RTO value instead of SLOW_RTO value. The trend is that the median FCT of "FastSlow" is higher than "Slow's" and the median FCT of FASOR is higher than "FastSlow's". This results from the fact that "FastSlow" and FASOR use fast retransmissions when probing for link errors, while actually the link is congested or bufferbloated. The FASOR initk1 variant is slightly more aggressive; with 400 clients and infinite buffer size, the median FCT of the initk1 variant is 0.42% and 0.33% longer than the initk4 variant in continuous and random cases, respectively.

The overhead of the token in FASOR+Token causes the median FCT of FASOR+Token to be 8.15% longer than "Slow's" with 400 clients and 2500B buffer size in the continuous cases. With the infinite buffer size, the accurate RTT estimation pays

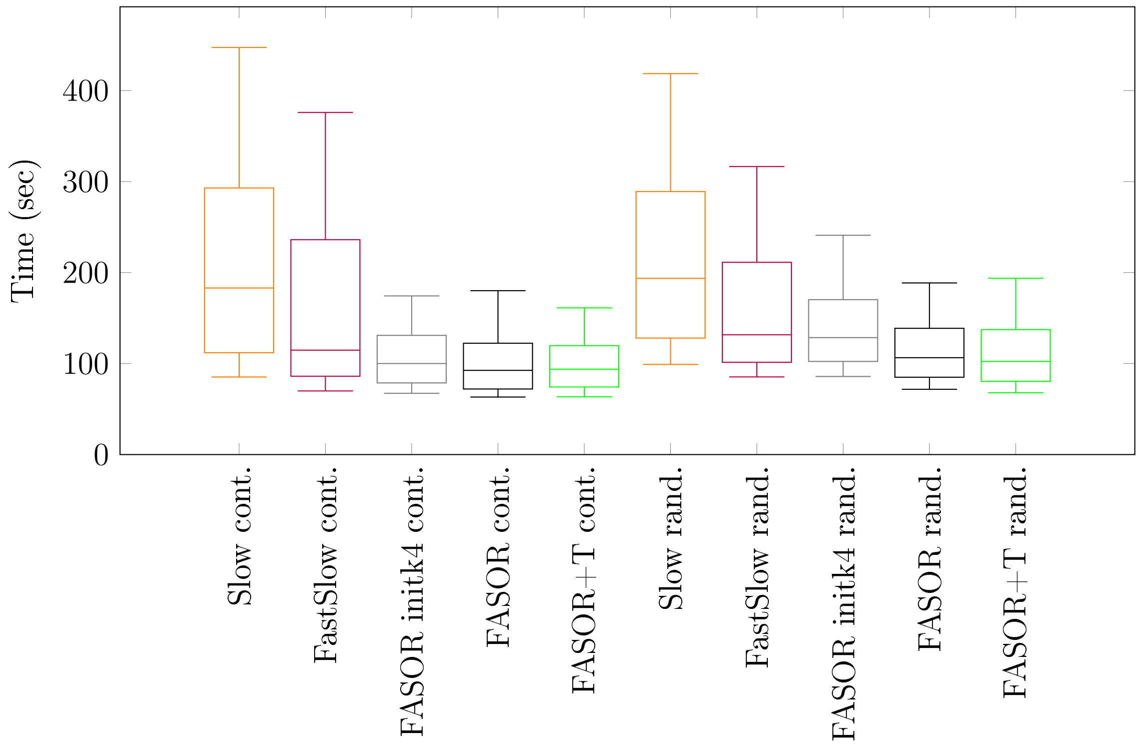

Figure 19: FCT of 10 clients, error-prone link, development algorithms

off and the median FCT of FASOR+Token is only 1.02% and 0.89% longer than that of "Slow" in continuous and random cases, respectively.

As expected, "Slow" performs poorly in the error-prone cases (see Figure 19). FASOR's backoff logic on the other hand performs well. The smaller initial K value improves performance by setting the RTO to a more realistic value; with high error rate the median FCT of the initk1 variant is 7.50% and 17.17% lower than initk4's in continuous and random cases, respectively.

8.2 Error-free link with continuous workload

The comparison to the baseline congestion control algorithms begins with the error-free link and continuous workload. The link is subject to congestion related losses only and to bufferbloat. The clients exchange 50 request-response pairs back to back. The CoAP over TCP clients have already established the connection and exchanged the CSM messages before the test runs.

Only one client exchanging the 50 messages in the network provides a baseline for comparisons. There are no losses, no queuing and no retransmissions. The median

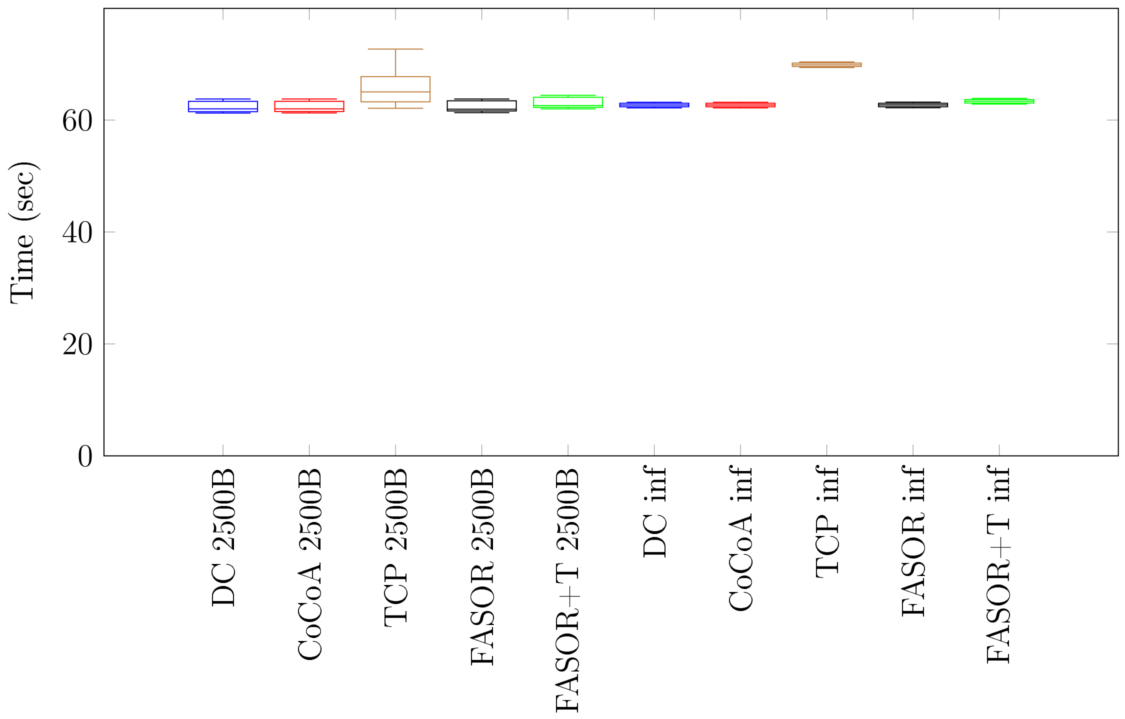

Figure 20: FCT of 50 continuous clients, error free link, using 2500B and infinite buffer sizes

flow completion time (FCT) is 33.003 seconds for DefaultCoAP, CoCoA and FASOR. The median FCT is 33.208 seconds for CoAP over TCP due to TCP's larger header size and 33.024 seconds for FASOR+Token due to the included one byte token. The RTT is roughly 660 ms.

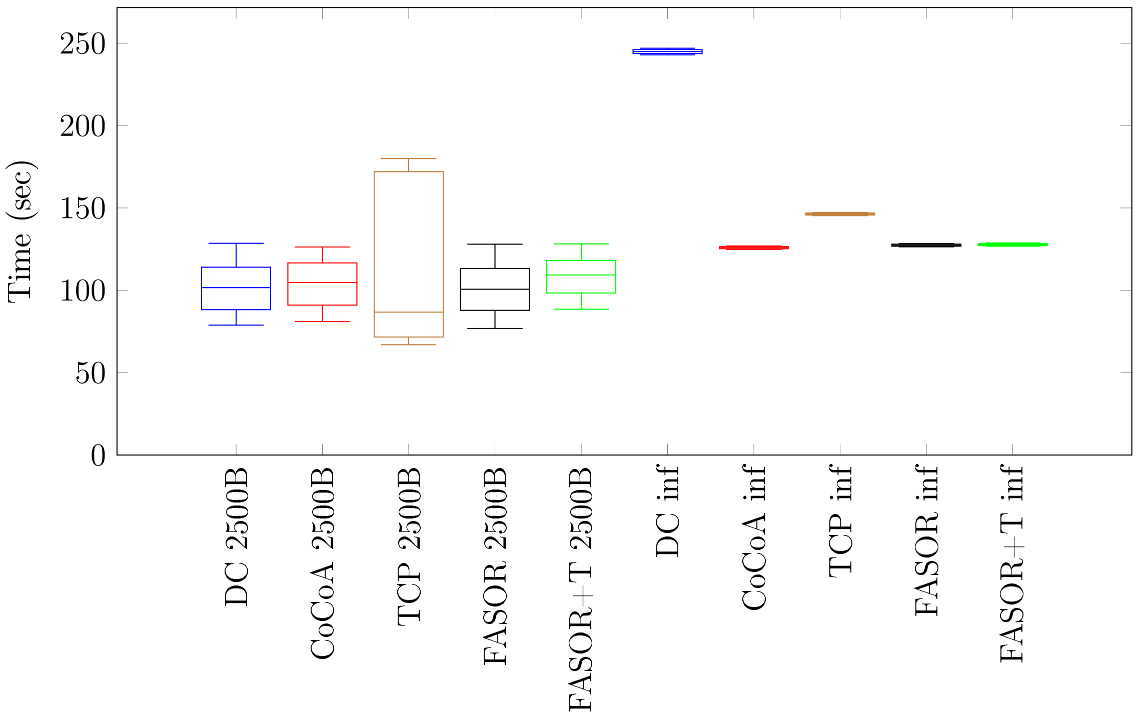

Figure 21: FCT of 100 continuous clients, error free link, using 2500B and infinite buffer sizes

With 50 clients, the median FCTs for DefaultCoAP, CoCoA and FASOR are roughly the same, 62 seconds using the small 2500B buffer size (see Figure 20) and 62.7 seconds using the infinite buffer. The median FCT of CoAP over TCP and FASOR+Token clients is slightly longer: 65.032 seconds (2500B buffer) and 69.868 seconds (infinite buffer) for CoAP over TCP and 62.690 seconds (2500B buffer) and 63.363 seconds (infinite buffer) for FASOR+Token. The higher FCT compared to the flow with only one client is mostly due to increased queuing delay (especially when using the infinite buffer), but also due to congestion related packet drops when using the small buffer size, which is visible from the higher percentiles. When using the small buffer, CoAP over TCP's notably longer median FCT (+3 seconds, about 5%) is due to the larger header causing longer serialization delay and more congestion in both uplink and downlink router buffers. With the infinite buffer, CoAP over TCP's 7.2 seconds (about 11%) longer median FCT is only due to longer queuing delay because of the larger header size [JPR+18].

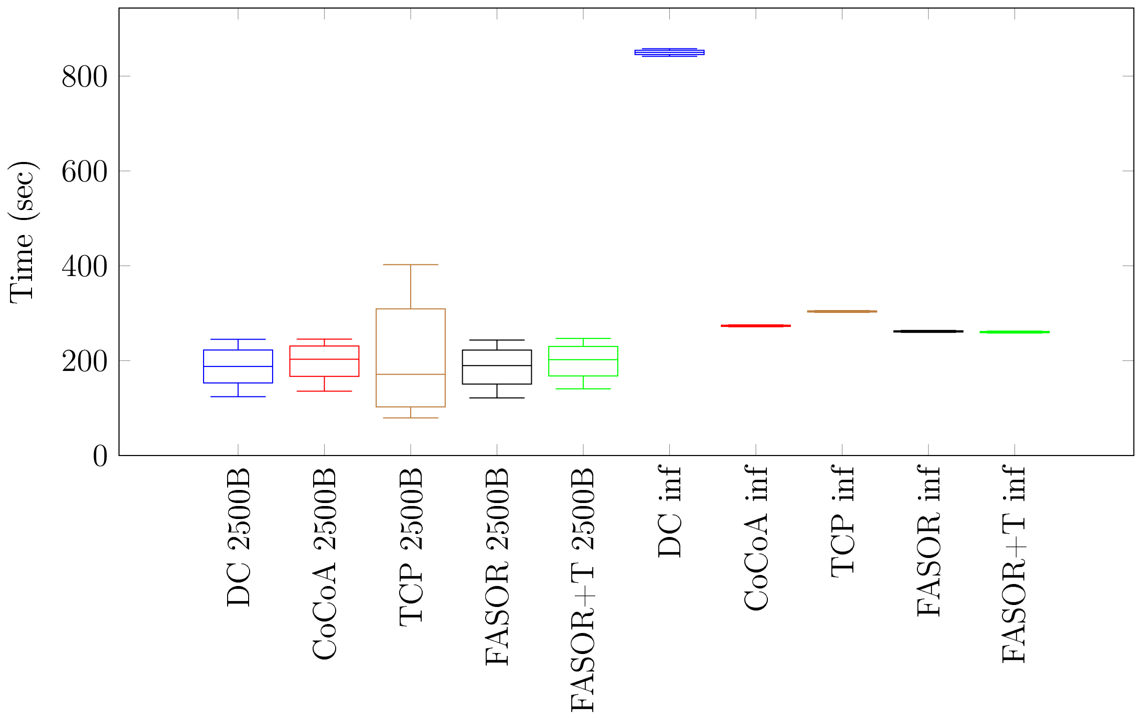

Figure 22: FCT of 200 continuous clients, error free link, using 2500B and infinite buffer sizes

With 100 clients, the congestion begins to increase with small buffer sizes resulting in more packet losses (see Figure 21). Using the small 2500B buffer size, the median FCT of DefaultCoAP is 101.6 seconds, CoCoA 104.7 seconds, CoAP over TCP 86.7 seconds, FASOR 100.642 seconds and FASOR+Token 109.303 seconds.

When using the infinite buffer, the median FCT of DefaultCoAP is 244.911 seconds, CoCoA 125.870 seconds, CoAP over TCP 146.308 seconds, FASOR 127.399 seconds and FASOR+Token 127.785 seconds. The prevailing RTT is higher than two seconds, causing DefaultCoAP to do unnecessary retransmissions on every new message. CoCoA, CoAP over TCP and FASORs have to unnecessarily retransmit only a couple of messages before the RTO value is set high enough.

With 200 clients the packet losses and queuing delay increases. With the 2500B buffer size, the median FCT of DefaultCoAP is 187.758 seconds, CoCoA 202.999 seconds, CoAP over TCP 171.142 seconds, FASOR 189.608 seconds and FASOR+Token 202.123 seconds (see Figure 22). CoCoA, CoAP over TCP and FASORs adjust the RTO values accordingly, requiring only a few unnecessary retransmissions in the beginning. The median FCT of FASOR is shorter than CoCoA's with all buffer sizes, and shorter than CoAP over TCP's with 28200B and infinite buffer sizes.

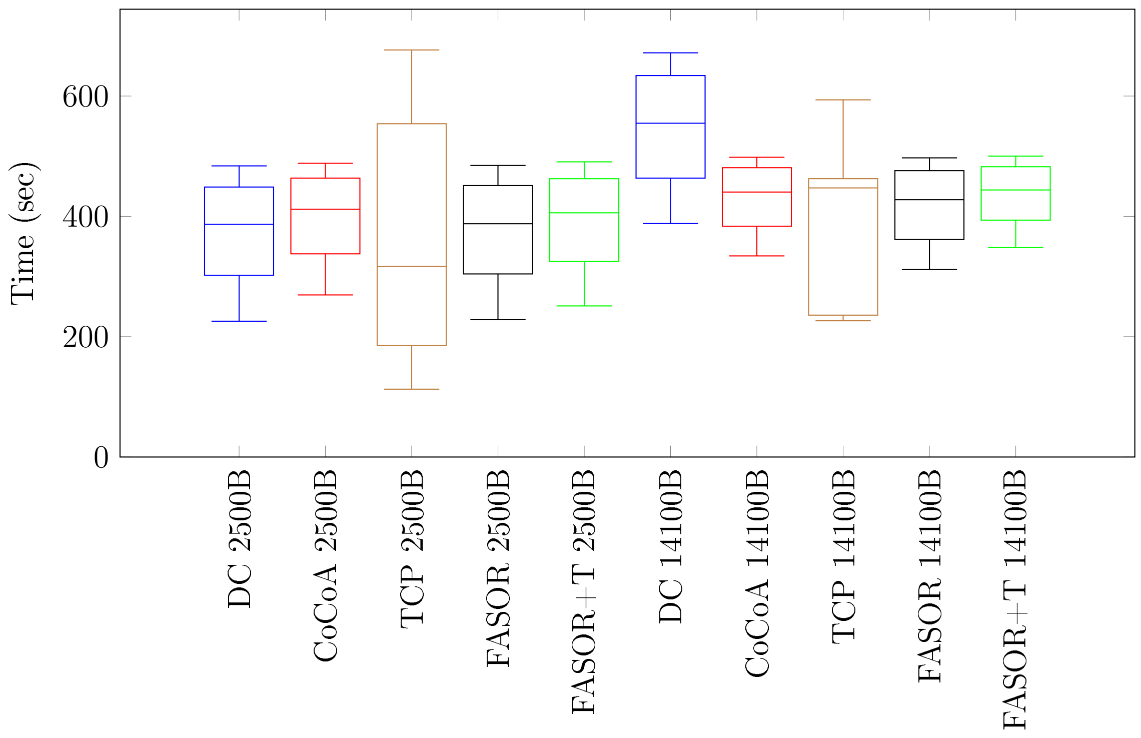

Figure 23: FCT of 400 continuous clients, error free link, using 2500B and 14100B buffer sizes

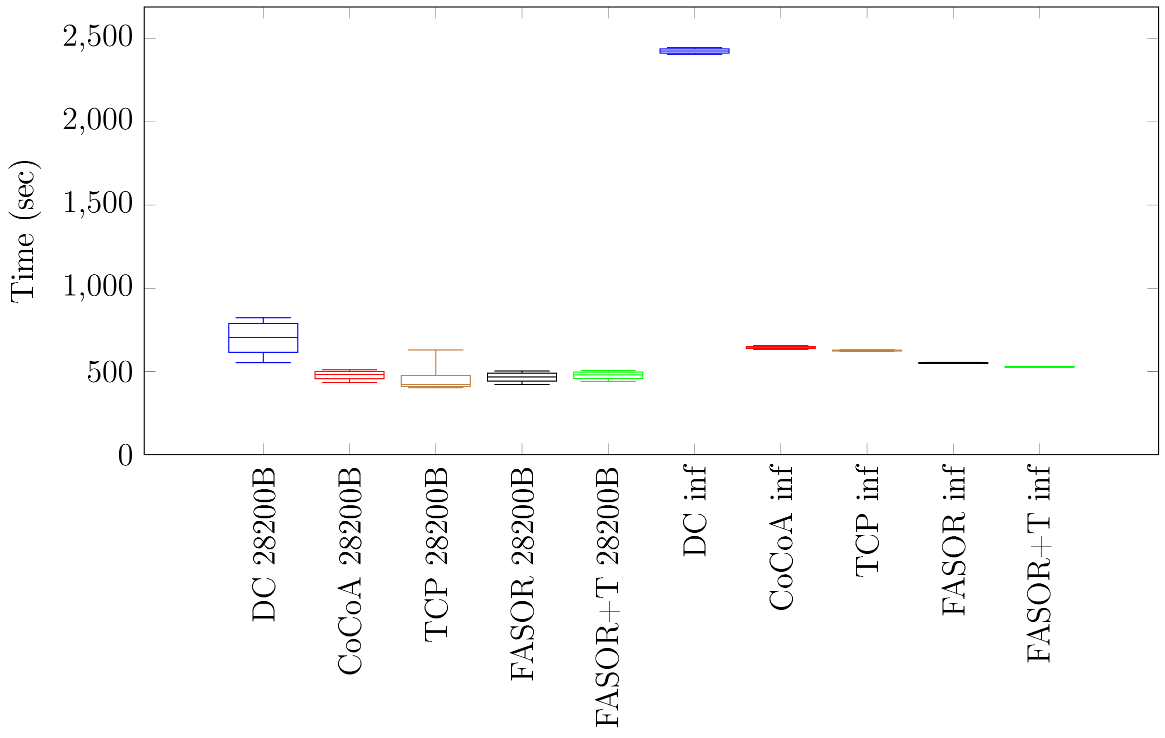

Figure 24: FCT of 400 continuous clients, error free link, using 28200B and infinite buffer sizes

With the infinite buffer size, the median FCT of DefaultCoAP is 849.945 seconds, CoCoA 273.486 seconds, CoAP over TCP 303.799 seconds, FASOR 261.704 seconds and FASOR+Token 260.251 seconds. DefaultCoAP starts to suffer a congestion collapse. When packets are not lost, TCP's larger header causes the median FCT to be longer than CoCoA's and FASORs' [JPR+18].

With 400 clients and the 2500B buffer size, the median FCT of CoAP over TCP is the shortest. The median FCT of DefaultCoAP is 386.756 seconds, CoCoA 411.899 seconds, CoAP over TCP 316.687 seconds, FASOR 387.750 seconds and FASOR+Token 405.912 seconds (see Figure 23). The median FCT of FASOR is 5.86% shorter than CoCoA's and roughly the same as DefaultCoAP's. FASOR+Token's RTT measurements lower the RTO causing more retransmissions [JRCK18b].

With the infinite buffer size all transmissions are queued and every unnecessary retransmission wastes the bottleneck link. The median FCT of DefaultCoAP is 2425.320 seconds, CoCoA 642.100 seconds, CoAP over TCP 626.073 seconds, FASOR 551.745 seconds and FASOR+Token 527.513 seconds (see Figure 24). DefaultCoAP suffers a congestion collapse while FASOR+Token and FASOR are the fastest ones to complete. The median FCT of FASOR is 77.25%, 14.07% and 11.87% shorter than DefaultCoAP's, CoCoA's and CoAP over TCP's respectively. The median FCT of FASOR+Token is even shorter: 4.39% shorter than FASOR's.

The problem with DefaultCoAP is that every new message has an initial RTO of 2-3 seconds while the prevailing RTT is much higher. In addition to RTT sampling, FASOR implements Karn's algorithm using the SLOW_RTO variable, which allows time for the queue to empty before retransmitting the message. FASOR+Token is able to calculate and adapt to the high prevailing RTT accurately using the additional information provided by the one-byte token.

8.3 Error-free link with random workload

With the random workload, the CoAP over TCP clients have to establish a new connection and exchange the Capabilities and Settings Message (CSM) at the beginning of each batch. With only one client exchanging the 50 messages, the median FCT of DefaultCoAP is 33.003 seconds, CoCoA 33.003 seconds, CoAP over TCP 47.093 seconds, FASOR 33.005 seconds and FASOR+Token 33.025 seconds.

With 50 random clients the results are similar to the continuous clients. The median FCT of DefaultCoAP is 62.002 seconds, CoCoA 62.002 seconds, CoAP over TCP

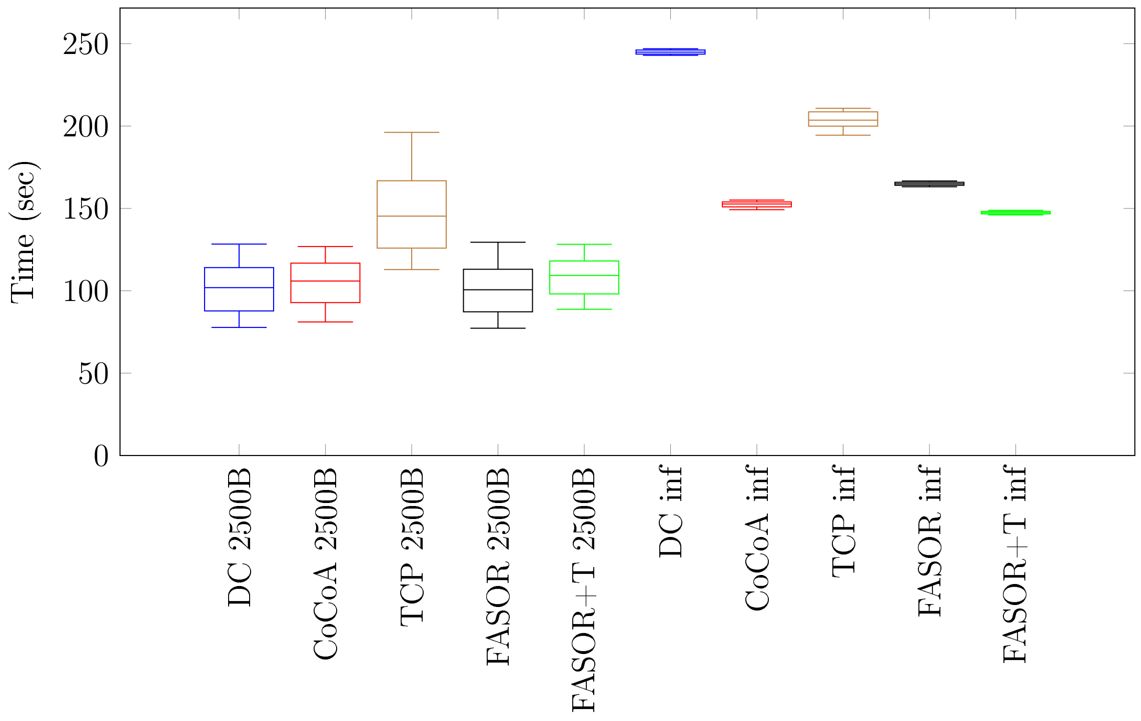

Figure 25: FCT of 100 random clients, error free link, using 2500B and infinite buffer sizes

88.433 seconds, FASOR 61.939 seconds and FASOR+Token 62.591 seconds.

With 100 random clients and the small 2500B buffer, the results of DefaultCoAP and CoCoA are similar to the results of the continuous clients (see Figure 25). The median FCT of DefaultCoAP is 101.908 seconds, CoCoA 105.918 seconds, CoAP over TCP 145.320 seconds, FASOR 100.617 seconds and FASOR+Token 109.293 seconds. The three-way handshakes of TCP make the FCT longer.

When using the infinite buffer size, where the RTT is above two seconds, CoCoA, CoAP over TCP and FASORs have longer median FCTs than the continuous clients because their state is reset after each batch. The initial RTO value is too low causing spurious RTOs. FASOR+Token is able to get unambiguous samples and adjusts the RTO quickly. The median FCT of FASOR+Token is only 15.37% longer than the continuous FCT while the median FCT of CoCoA is 21.31% longer than the continuous FCT. The median FCT of DefaultCoAP is 244.863 seconds, CoCoA 152.693 seconds, CoAP over TCP 203.548 seconds, FASOR 164.925 seconds and FASOR+Token 147.420 seconds.

With 200 random clients and the larger buffer sizes every new CoCoA batch starts with a too low RTT estimate and the weak samples do not adjust the RTT estimate

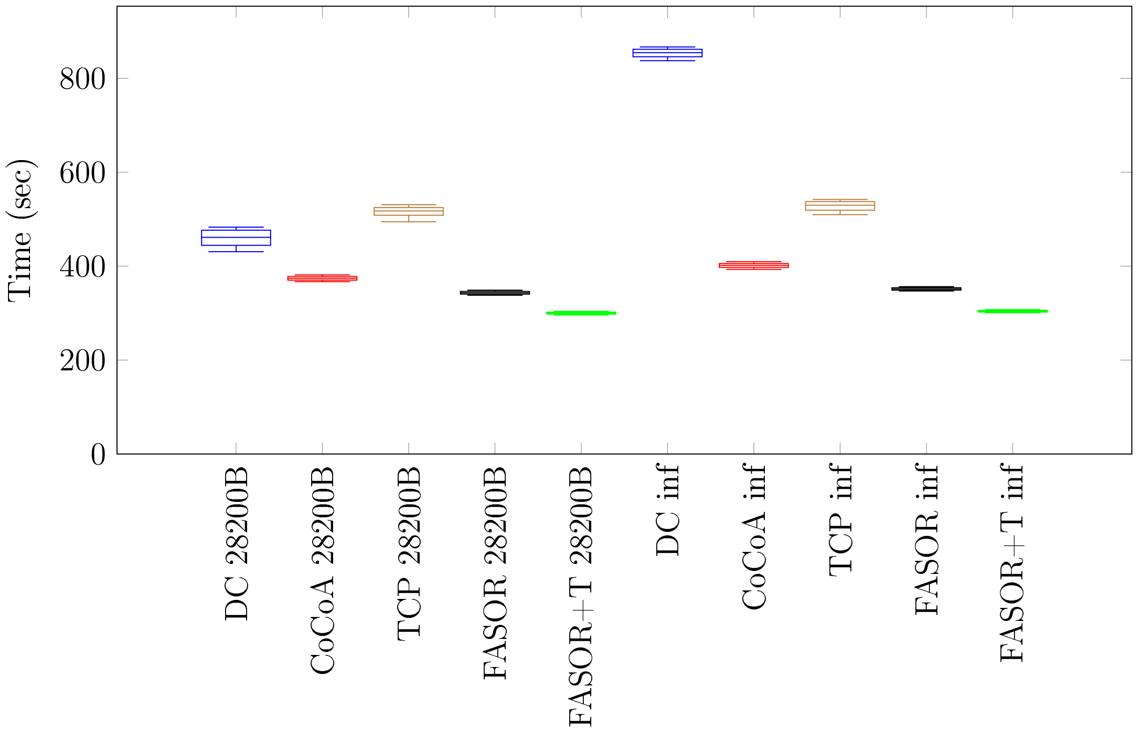

Figure 26: FCT of 200 random clients, error free link, using 28200B and infinite buffer sizes

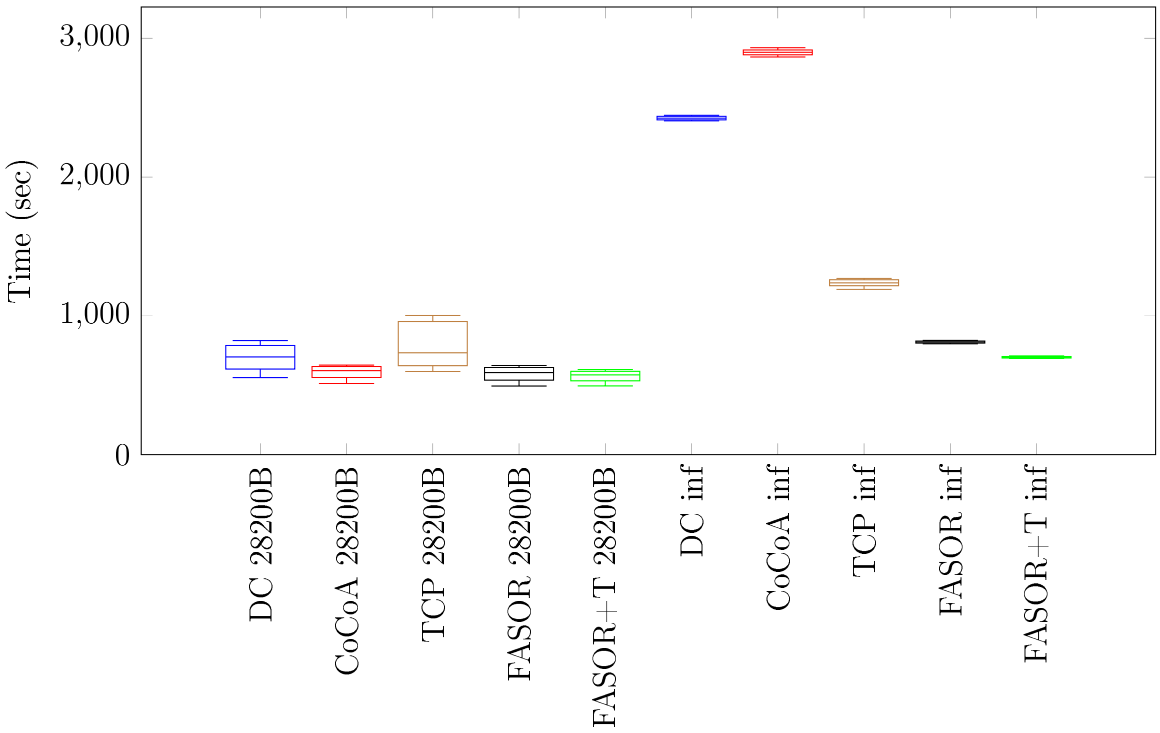

Figure 27: FCT of 400 random clients, error free link, using 28200B and infinite buffer sizes

quickly enough. With the 28200B buffer size, the median FCT of DefaultCoAP is 461.338 seconds, CoCoA 374.131 seconds, CoAP over TCP 517.621 seconds, FASOR 343.212 seconds and FASOR+Token 300.028 seconds (see Figure 26). The median FCT of FASOR is 8.27% shorter than CoCoA's and the median FCT of FASOR+Token is 12.58% shorter than FASOR's.

With 400 random clients the trend is similar as with 200 random clients. With the 2500 buffer size, the median FCT of DefaultCoAP is 385.086 seconds, CoCoA 417.007 seconds, CoAP over TCP 460.947 seconds, FASOR 391.240 seconds and FASOR+Token 409.227 seconds. The median FCT of FASOR is 6.18% shorter than CoCoA's while CoAP over TCP is the slowest one to complete.

With the 28200B buffer size the queue is long enough that the initial RTO is too small. All algorithms do spurious RTOs, but FASOR+Token acquires a sample the fastest at the beginning of each batch (see Figure 27). The median FCT of FASOR+Token is 2.63% and 4.91% shorter than FASOR's and CoCoA's respectively.

With the infinite buffer size, the median FCT of DefaultCoAP is 2424.610 seconds, CoCoA 2898.480 seconds, CoAP over TCP 1237.885 seconds, FASOR 812.070 seconds and FASOR+Token 702.802 seconds. Both DefaultCoAP and CoCoA experience a congestion collapse while the FASORs are congestion safe. The median FCT of FASOR is 72.03% shorter than CoCoA's and the median FCT of FASOR+Token is 13.46% shorter than FASOR's.

Again, FASOR implements the Karn's algorithm and is able to get unambiguous samples after a few retransmissions. FASOR+Token is able to convert otherwise ambiguous samples into unambiguous to adjust the RTO value quickly.

8.4 Error-prone link with continuous workload

In the error-prone test cases, packet losses are due to link errors and not congestion. For baseline, without errors and with 10 clients, the median FCT of DefaultCoAP is 33.220 seconds, CoCoA 33.220 seconds, CoAP over TCP 33.440 seconds, FASOR 33.221 seconds and FASOR+Token 33.242 seconds.

With low error rate the median FCT of DefaultCoAP is 38.126 seconds, CoCoA 35.531 seconds, CoAP over TCP 35.341 seconds, FASOR 35.618 seconds and FASOR+Token 35.758 seconds. When the error rate increases, the FCT grows. With medium error rate the median FCT of DefaultCoAP is 70.212 seconds, CoCoA 57.002 seconds, CoAP over TCP 50.726 seconds, FASOR 53.253 seconds and FA-

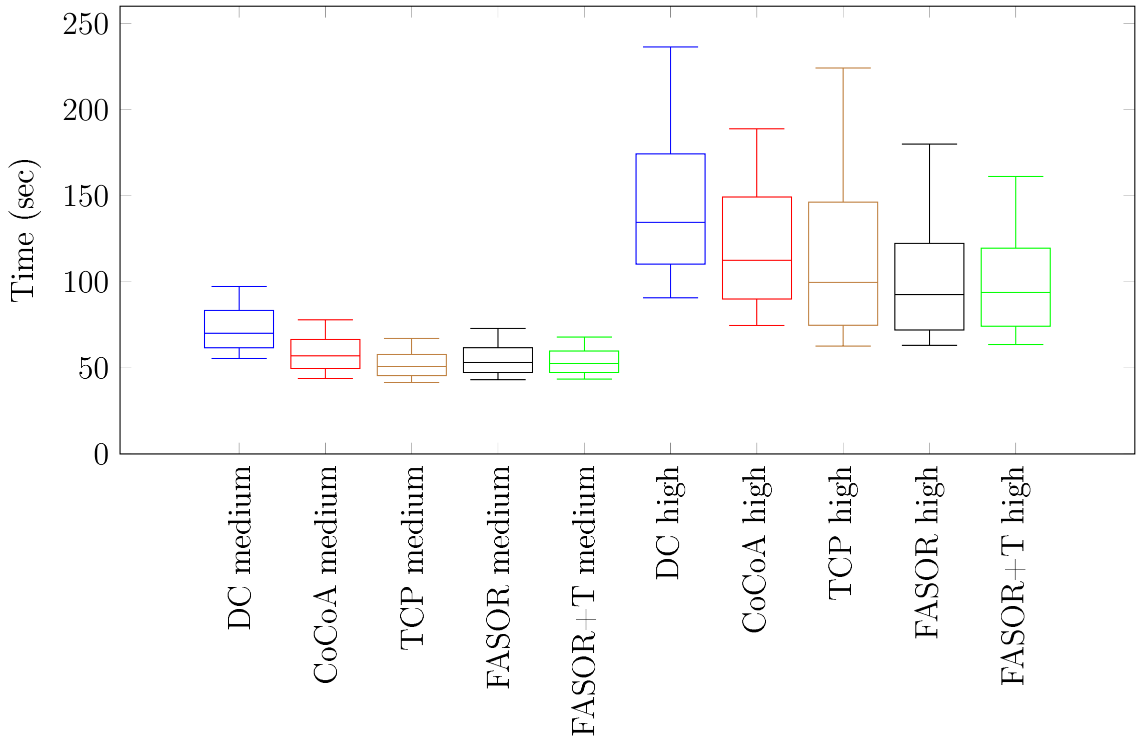

Figure 28: FCT of 10 continuous clients, error-prone link, using medium and high error rates

SOR+Token 52.598 seconds (see Figure 28).

As the RTO expiration is the only loss recovery method, accurate RTT estimation is important. FASOR is able to perform better than DefaultCoAP and CoCoA even though they have unfair advantage in error-prone cases (not being congestion safe in error free cases) [JRCK18b].

With medium error rate the median FCT of FASOR is 24.15% shorter than DefaultCoAP's and 6.58% shorter than CoCoA's. The median FCT of FASOR+Token is 1.23% shorter than FASOR's.

With high error rate the median FCT of DefaultCoAP is 134.596 seconds, CoCoA 112.606 seconds, CoAP over TCP 99.721 seconds, FASOR 92.536 seconds and FASOR+Token 93.822 seconds. The median FCT of FASOR is 31.25% shorter than DefaultCoAP's, 17.82% shorter than CoCoA's and 7.21% shorter than CoAP over TCP's.

The novel backoff logic allows FASOR clients to recover from the packet losses quickly while still being congestion safe.

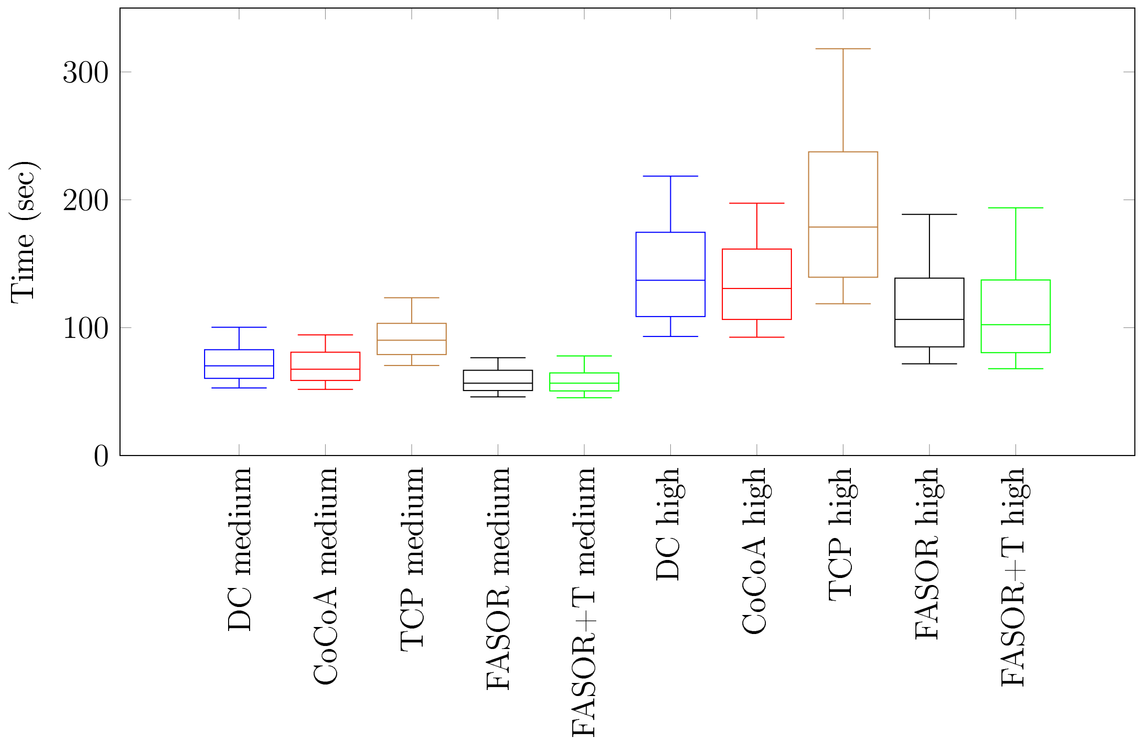

Figure 29: FCT of 10 random clients, error-prone link, using medium and high error rates

8.5 Error-prone link with random workload

Without errors and with 10 clients the median FCT of DefaultCoAP is 33.220 seconds, CoCoA 33.220 seconds, CoAP over TCP 47.598 seconds, FASOR 33.222 seconds and FASOR+Token 33.243 seconds. With low error rate the median FCT of DefaultCoAP is 38.260 seconds, CoCoA 37.193 seconds, CoAP over TCP 52.015 seconds, FASOR 35.954 seconds and FASOR+Token 35.947 seconds.

CoAP over TCP's median FCT is the longest due to the TCP connection establishment segments as with the continuous clients. These segments are also subject to packet losses. [JPR+18]

With medium error rate the median FCT of DefaultCoAP is 70.148 seconds, CoCoA 67.484 seconds, CoAP over TCP 90.179 seconds, FASOR 56.564 seconds and FASOR+Token 56.580 seconds (see Figure 29). The median FCT of FASOR is 19.44% shorter than DefaultCoAP's and 16.18% shorter than CoCoA's.

With high error rate the median FCT of DefaultCoAP is 137.065 seconds, CoCoA 130.645 seconds, CoAP over TCP 178.696 seconds, FASOR 106.445 seconds and

FASOR+Token 102.332 seconds. The median FCT of FASOR is 22.34% shorter than DefaultCoAP's and 18.52% shorter than CoCoA's. The median FCT of FASOR+Token is 3.86% shorter than FASOR's.

As with the continuous clients, FASOR's fast retransmit allows quick error recovery in most cases. FASOR+Token is able to measure the prevailing RTT quickly and accurately to recover from packet-errors more efficiently. Since the links used in IoT traffic are often error-prone, this is an important improvement.

8.6 Summary

FASOR is able to perform well in the difficult scenarios, where DefaultCoAP and CoCoA experience a congestion collapse. FASOR implements Karn's algorithm using the SLOW_RTO variable and is even faster to complete than CoAP over TCP when facing bufferbloat.

FASOR with accurate RTT measurements adjusts the RTO quickly, which is especially useful with the random workload. FASOR+Token has the shortest FCT in bufferbloat scenarios in addition to error-prone scenarios.

Both FASOR variants have shorter FCTs in the error-prone cases than DefaultCoAP and CoCoA, even though DefaultCoAP and CoCoA have unfair advantage of being too aggressive in error-free cases.

9 Use cases and configuration

CoAP has many advantages over the other protocols briefly discussed in Section 2.4. From a practical viewpoint CoAP integrates seamlessly with the current TCP/IP protocol stack and Internet infrastructure [KRS14]. CoAP provides interoperability into complex Wireless Sensor Network architectures that typically include application-dependent and engineering-oriented sensors. The HTTP<->CoAP translations specified in [SHB14] allow accessing CoAP nodes easily from, for example, traditional web browsers.

The RESTful architecture provides a gentle learning curve for programmers that have previous experience in web software development. The use of UDP allows multicasting and the use of non-reliable messaging if the application does not require reliability, for example sending temperature measurements as non-confirmable messages and temperature alerts as confirmable messages.

In the spirit of this thesis, CoAP allows tuning the transmission parameters and even replacing the congestion control algorithm. For example, if the deployment environment is known to be bufferbloated, the default RTO of 2-3 seconds can be increased to avoid spurious RTOs.

To ensure the stability of the Internet, a congestion control algorithm must be sufficiently responsive to congestion [BCC+98]. For an algorithm to be safe to use in the Internet, it should be usable in all situations. As per the results of Section 8, if CoAP is used over UDP as it is recommended, then a FASOR variant is most suitable. Because some networks and network devices block UDP traffic, CoAP over TCP is a viable option.

In some situations, where the deployment scenario and its properties are known, it is possible to do optimizations. Two special scenarios are considered as examples at the end of this Chapter.

Ideally, the quality of the path could be measured to select a best suited congestion control algorithm. As discussed in Section 5.1, packet drops due to congestion and link errors are indistinguishable from each other. For example, the Point-to-Point Protocol (PPP) has a mechanism for link quality monitoring, the Link Control Protocol [Sim96]. The protocol sends Link-Quality-Report packets containing the number of packets and octets that were transmitted and successfully received. The receiver compares the values to the previous report, giving an indication of the current link quality. The implementation can then decide an action based on the reports. This approach works only for one link and not for the end-to-end path, as it does not distinguish the cause of the packet drops and thus cannot supply useful information for congestion control algorithm selection.

The IEEE 802.15.4 LR-WPAN standard, used by ZigBee among others, has a Link Quality Indication mechanism on the physical layer [Erg04]. The Link Quality Indication result is an integer from 0x00 (the lowest quality) to 0xff (the highest quality). It is measured from an estimate of received signal power and/or signal-to-noise ratio. The network and application layers can use the result. An application could, for example, initially use FASOR without a token to conserve bandwidth, but start including the token when the link quality deteriorates below some threshold.

TCP can utilize ECN as a limited additional information about the presence of congestion. TCP can also act as a fall back protocol; if CoAP over UDP does not receive acknowledgements, one may try switching to CoAP over TCP.

As the IoT devices are small and constrained, it may not be feasible to include support for multiple congestion control algorithms. The software and libraries loaded in the devices can be configured before deployment and any excess algorithms be removed on compile time to limit runtime overhead and required storage space.

The most constrained devices, Class 0 devices [BEK14], have less than 10 kilobytes of RAM. Those devices may require the use of DefaultCoAP as it has the smallest memory footprint. DefaultCoAP requires 2 bytes of RAM per client, CoCoA 29 bytes and Linux RTO 21 bytes [BGDP16]. FASOR requires roughly the same amount of RAM as Linux RTO. The required variables are current RTO value, RTT variation, smoothed RTT, retransmission count, send timestamp, current state and SLOW_RTO value. The use of FASOR with accurate RTT measurements requires more memory as the ordinal retransmission number and respective transmission timestamps have to be stored in memory. The amount of additional memory required by the congestion control algorithms is negligible compared with other CoAP features, such as the Datagram Transport Layer Security (DTLS) that consumes about 2 kilobytes of RAM [BGDP16].

The first example scenario is a cargo container ship. There are one thousand containers onboard, each one fitted with measurement devices for location and temperature. The devices connect to an on-board base station using a short range protocol (such as WiFi) and the base station connects to the Internet using a satellite link. The propagation delay of a Geostationary Orbit satellite link alone is 239.6 - 279.0 milliseconds, increasing the RTT by 479.2 - 558 milliseconds [AGS99]. The metallic containers and their contents obstruct the radio signals causing poor signal reception and lost packets. Rain hinders the quality of the satellite link.

For this kind of situation the best performing congestion control algorithm as per the results is FASOR+Token. It can effectively adapt to the large RTT and it performs the best in error-prone cases. Even though the token increases the overhead, it is much less than the overhead of the TCP header. On compile time the algorithm could be configured to use initial K of default 4 instead of 1 because the RTT variance can be high in this case. If low energy consumption is desired, the slow RTO factor can be increased to minimize the amount of spurious RTOs, but with a performance penalty.

The second example scenario is a building automation case, where temperature, humidity, motion and light sensors are scattered all around a large building. There are also actuators, such as lights and door locks. The device installations are fixed and both communicate and are powered by wired Ethernet. Even though the devices have an ample supply of power through Power-over-Ethernet, the devices are still constrained by their manufacturing cost and physical size. The devices are connected to CoAP gateways that provide security features. The gateways are also connected to the internet via wired Ethernet.

The link is mostly error free. Message bursts and congestion occur, for example, when all the lights are lit simultaneously and the light sensors react to it. The system is managed by network administrators who do not impose artificial limits on UDP traffic and thus TCP fall back is not needed. CoAP over TCP performs well in error free cases, but the connection establishment hinders the random workload. Again, FASOR+Token has the best general performance, but the development prototype "Slow" algorithm would perform better in cases with only congestion related losses.

10 Conclusion and future work

The Internet of Things (IoT) consists of constrained devices, such as thermometers, that are often battery powered and have low amounts of memory and processing power. The devices communicate using networks that also have constraints, such as low bit rate and high error rate. The Constrained Application Protocol (CoAP) is a low overhead web transfer protocol for IoT communication. The protocol has similarities with HTTP and uses a REST architecture with a client/server model. CoAP is built on top of UDP and supports reliable messaging. CoAP includes a very simple congestion control algorithm called DefaultCoAP in this thesis.

If the path between the client and the server has plenty of available bandwidth and no link errors, the congestion control algorithm makes little difference. This is often not the case in IoT networks. CoAP Simple Congestion Control/Advanced (CoCoA) is a congestion control algorithm specifically made for CoAP and IoT communications to combat the shortcomings of DefaultCoAP. Unlike DefaultCoAP, CoCoA uses RTT measurements to estimate a proper RTO value. The empirical performance evaluation also included the use of CoAP over TCP using the general purpose Linux RTO congestion control algorithm.

The empirical testing was performed in a controlled network that emulated a typical IoT network. Both the clients and server used actual implementations of the algorithms implemented into libcoap, an open-source CoAP library for C. The test scenarios had varying amounts of clients (up to 400), bufferbloat and errors. The results reveal that both DefaultCoAP and CoCoA experience a congestion collapse in certain challenging cases, while CoAP over TCP performs sub optimally in error prone cases.

A new congestion control algorithm, Fast-Slow RTO (FASOR) [JRCK18b, JKRC18], has been proposed as a solution. FASOR introduces a novel backoff logic allowing fast recovery from link errors while being congestion safe. FASOR can be complemented with a token value or a CoAP option to remove retransmission ambiguity.

The results indicate that FASOR performs well in IoT environments and is able to outperform the competition while being congestion safe in the cases where DefaultCoAP and CoCoA were not. FASOR+Token outperforms plain FASOR in the demanding error-prone random workload case, which emulates short message exchanges typical for IoT communications.

The vast amount of IoT devices connected to the Internet signifies that congestion control even in seemingly small IoT communications is important to ensure the stability of the Internet. A congestion control algorithm should be safe to use in all scenarios. The algorithms can be optimized using configuration if the deployment scenario and its properties are known.

Future work includes potentially developing an aging mechanism for FASOR and tests with TCP related techniques, such as enabling ECN as suggested in [GCS19]. Besides ECN, Explicit Congestion Notification, also Explicit Loss Notification has been proposed, for example in [DJ01], but not standardized.

The TCP implementation used in the tests was a full-fledged one while lightweight TCP implementations for constrained devices exist. Although the FASOR token and option are influenced by the TCP timestamp option (12 bytes), the use of the timestamp option with TCP was not included in the tests.

More workloads. A workload that generates requests at a constant rate, for example temperature measurements every 30 seconds, was created but not included in the tests. The tests of [JDK15] included Competing and Mixed workloads, where different congestion control algorithms were run in the same test scenario.

The error prone cases were subject to link error related losses only. Further testing could include cases with larger amount of clients for both congestion and link error related losses. Related to link error cases, the use of Forward Error Correction could be investigated.

Although constrained devices might not be the ideal target group, the possibility of exploiting late arriving acknowledgements for additional RTT samples could be investigated.

References

AFGM+15 Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M. and Ayyash, M., Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, volume 17, June 2015, pages 2347–2376.

AGS99 Allman, M., Glover, D. and Sanchez, L., Enhancing TCP Over Satellite Channels using Standard Mechanisms. RFC 2488, January 1999.

AMQP AMQP, Advanced Message Queuing Protocol. URL `https://www.amqp.org`. Accessed 21.4.2019.

APB09 Allman, M., Paxson, V. and Blanton, E., TCP Congestion Control. RFC 5681, September 2009.

AVTP+17 Adelantado, F., Vilajosana, X., Tuset-Peiro, P., Martinez, B., Melia-Segui, J. and Watteyne, T., Understanding the Limits of LoRaWAN. *IEEE Communications Magazine*, volume 55, September 2017, pages 34–40.

BBGD17 Bormann, C., Betzler, A., Gomez, C. and Demirkol, I., CoAP Simple Congestion Control/Advanced. Internet Draft, March 2017. Work in progress.

BBJS14 Borman, D., Braden, B., Jacobson, V. and Scheffenegger, R., TCP Extensions for High Performance. RFC 7323, September 2014.

BCC+98 Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J. and Zhang, L., Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309, April 1998.

BEK14 Bormann, C., Ersue, M. and Keranen, A., Terminology for Constrained-Node Networks. RFC 7228, May 2014.

BGDP16 Betzler, A., Gomez, C., Demirkol, I. and Paradells, J., CoAP Congestion Control for the Internet of Things. *IEEE Communications Magazine*, number 7, July 2016, pages 154–160.

BLT$^+$18 Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B. and Raymor, B., CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets, RFC 8323, February 2018.

Bluetooth Bluetooth SIG. URL `https://www.bluetooth.com`. Accessed 21.4.2019.

BS16 Bormann, C. and Shelby, Z., Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, August 2016.

CCRJ14 Cheng, Y., Chu, J., Radhakrishnan, S. and Jain, A., TCP Fast Open. RFC 7413, December 2014.

CMHH17 Chen, M., Miao, Y., Hao, Y. and Hwang, K., Narrow Band Internet of Things. *IEEE Access*, volume 5, 2017, pages 20557–20577.

DDS DDS, Data Distribution Service. URL `https://www.omg.org/omg-dds-portal`. Accessed 21.4.2019.

DGV04 Dunkels, A., Gronvall, B. and Voigt, T., Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. *29th Annual IEEE International Conference on Local Computer Networks*, Tampa, Florida, USA, November 2004, pages 455–462.

DJ01 Ding, W. and Jamalipour, A., A New Explicit Loss Notification with Acknowledgment for Wireless TCP. *12th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications. PIMRC 2001. Proceedings*, San Diego, California, USA, October 2001.

DMK$^+$01 Dawkins, S., Montenegro, G., Kojo, M., Magret, V. and Vaidya, N., End-to-end Performance Implications of Links with Errors. RFC 3155, August 2001.

Dun03 Dunkels, A., Full TCP/IP for 8-bit Architectures. *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, Stanford, California, USA, May 2003, ACM, pages 85–98.

EC-GSM EC-GSM-IoT, Extended Coverage GSM Internet of Things. URL `https://www.gsma.com/iot/extended-coverage-gsm-internet-of-things-ec-gsm-iot`. Accessed 21.4.2019.

EF08 Eggert, L. and Fairhurst, G., Unicast UDP Usage Guidelines for Application Designers. RFC 5405, November 2008.

eMTC-NB eMTC, enhanced Machine Type Communication, and NB-IoT, Narrow Band IoT - Standards for the IoT. URL `http://www.3gpp.org/news-events/3gpp-news/1805-iot_r14`. Accessed 21.4.2019.

Erg04 Ergen, S. C., ZigBee/IEEE 802.15.4 Summary. *UC Berkeley*, September 2004, URL `http://pages.cs.wisc.edu/~suman/courses/707/papers/zigbee.pdf`. Accessed 21.4.2019.

GAMC18 Gomez, C., Arcia-Moret, A. and Crowcroft, J., TCP in the Internet of Things: From Ostracism to Prominence. *IEEE Internet Computing*, volume 22, January 2018, pages 29–41.

GCS19 Gomez, C., Crowcroft, J. and Scharf, M., TCP Usage Guidance in the Internet of Things (IoT). Internet-draft, March 2019. Work in Progress.

GN11 Gettys, J. and Nichols, K., Bufferbloat: Dark Buffers in the Internet. *ACM Queue*, volume 9, November 2011.

GRM$^+$10 Ghosh, A., Ratasuk, R., Mondal, B., Mangalvedhe, N. and Thomas, T., LTE-Advanced: Next-Generation Wireless Broadband Technology. *IEEE Wireless Communications*, volume 17, June 2010, pages 10–22.

Har15 Hartke, K., Observing Resources in the Constrained Application Protocol (CoAP). RFC 7641, September 2015.

HFGN12 Henderson, T., Floyd, S., Gurtov, A. and Nishida, Y., The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, April 2012.

IML$^+$03 Inamura, H., Montenegro, G., Ludwig, R., Gurtov, A. and Khafizov, F., TCP over Second (2.5G) and Third (3G) Generation Wireless Networks. RFC 3481, February 2003.

Ingenu Ingenu. URL `https://www.ingenu.com`. Accessed 21.4.2019.

IS18 Iyengar, J. and Swett, I., QUIC Loss Detection and Congestion Control. Internet-draft, December 2018. Work in Progress.

Jac88 Jacobson, V., Congestion Avoidance and Control. *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, Stanford, California, USA, August 1988, ACM, pages 314–329.

JDK15 Järvinen, I., Daniel, L. and Kojo, M., Experimental Evaluation of Alternative Congestion Control Algorithms for Constrained Application Protocol (CoAP). *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Milan, Italy, December 2015, pages 453–458.

JKRC18 Järvinen, I., Kojo, M., Raitahila, I. and Cao, Z., Fast-Slow Retransmission Timeout and Congestion Control Algorithm for CoAP. Internet-draft, October 2018. Work in Progress.

JPR$^+$18 Järvinen, I., Pesola, L., Raitahila, I., Cao, Z. and Kojo, M., Performance Evaluation of Constrained Application Protocol over TCP. *2018 IEEE 88th Vehicular Technology Conference*, Chicago, Illinois, USA, August 2018, IEEE.

JRCK18a Järvinen, I., Raitahila, I., Cao, Z. and Kojo, M., Is CoAP Congestion Safe? *Proceedings of the Applied Networking Research Workshop*, ANRW '18, New York City, New York, USA, July 2018, pages 43–49.

JRCK18b Järvinen, I., Raitahila, I., Cao, Z. and Kojo, M., FASOR Retransmission Timeout and Congestion Control Mechanism for CoAP. *2018 IEEE Global Communications Conference*, Abu Dhabi, UAE, December 2018.

KKJ19 Koster, M., Keränen, A. and Jimenez, J., Publish-Subscribe Broker for the Constrained Application Protocol (CoAP). Internet-draft, March 2019. Work in Progress.

KP87 Karn, P. and Partridge, C., Improving Round-trip Time Estimates in Reliable Transport Protocols. *SIGCOMM'87 Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology*, Stowe, Vermont, USA, August 1987, pages 2–7.

KRS14 Khattak, H. A., Ruta, M. and Sciascio, E. E. D., CoAP-based Healthcare Sensor Networks: a survey. *Proceedings of 2014 11th International*

Bhurban Conference on Applied Sciences Technology (IBCAST) Islamabad, Pakistan, 14th - 18th January, 2014, Jan 2014, pages 499–503.

libcoap libcoap: C-Implementation of CoAP. URL `https://libcoap.net/`. Accessed 21.4.2019.

LMP+05 Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E. et al., TinyOS: An Operating System for Sensor Networks. In *Ambient Intelligence*, Springer, 2005, pages 115–148.

LoRaWAN About LoRaWAN. URL `https://lora-alliance.org/about-lorawan`. Accessed 21.4.2019.

MQTT MQTT, Message Queuing Telemetry Transport. URL `https://mqtt.org`. Accessed 21.4.2019.

Nag84 Nagle, J., Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

NFC NFC Forum. URL `https://nfc-forum.org`. Accessed 21.4.2019.

PACS11 Paxson, V., Allman, M., Chu, J. and Sargent, M., Computing TCP's Retransmission Timer. RFC 6298, June 2011.

Pes19 Pesola, L., An Experimental Evaluation of Constrained Application Protocol Performance over TCP. Master's thesis, University of Helsinki, 2019. Work in Progress.

RFB01 Ramakrishnan, K., Floyd, S. and Black, D., The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001.

SHB14 Shelby, Z., Hartke, K. and Bormann, C., The Constrained Application Protocol (CoAP). RFC 7252, June 2014.

Sigfox Sigfox. URL `https://www.sigfox.com/`. Accessed 21.4.2019.

Sim96 Simpson, W., PPP Link Quality Monitoring. RFC 1989, August 1996.

Ste97 Stevens, W., TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, January 1997.

Tou97 Touch, J., TCP Control Block Interdependence. RFC 2140, April 1997.

Weightless	Weightless SIG. URL `http://www.weightless.org`. Accessed 21.4.2019.
Wi-Fi	Wi-Fi Alliance. URL `http://www.wi-fi.org`. Accessed 21.4.2019.
XMPP	XMPP, Extensible Messaging and Presence Protocol. URL `https://www.xmpp.org`. Accessed 21.4.2019.
Z-Wave	Z-Wave Alliance. URL `https://z-wavealliance.org`. Accessed 21.4.2019.