Master's Thesis

A Resilient Transport Layer for Messaging Systems

September 12, 2007

David Fuchs
fuchsd@student.ethz.ch

Supervision:
Dr. Sean Rooney (IBM Research GmbH, Zurich Research Laboratory)
Prof. Dr. Gustavo Alonso (ETH Zurich, Institute for Pervasive Computing)

Abstract

MQTT, the Message Queue Telemetry Transport protocol developed by IBM, is a lightweight publish/subscribe protocol for network edge devices. MQTT typically runs over a TCP connection. While TCP provides reliable and in-order delivery for a connection, it cannot deal with failures such as a complete disconnect of two peers or peers failing due to hardware errors or power losses. Usually, it is the application's task to detect and correct such problems.

This paper presents ReTCP, a general purpose, light-weight addition to TCP-based network stacks. ReTCP provides reliable and in-order packet delivery for an application, even in the presence of network failures causing the underlying TCP connection to disconnect, or application failures due to power losses etc. ReTCP is explained and implemented in the context of MQTT TCP-based network stack.

The implementation is tested and its performance compared to the one of the existing stack under several scenarios.

Keywords: MQTT, messaging systems, failure resiliency, persistency layer, TCP, ReTCP

Contents

List of Figures

Chapter 1

Introduction

In this first chapter, a short introduction of MQTT is presented, and the motivation and goals of ReTCP are explained. The rest of the paper is then structured as follows:

Chapter 2 explains the MQTT protocol in more detail, and then defines the design of ReTCP and the expected advantages of using a ReTCP stack.

Chapter 3 describes the architecture of the MQTT networking stack, and how ReTCP is implemented within this architecture.

Chapter 4 shows the results of the unit-, integration-, and performance-tests that were executed.

Chapter 5 concludes the paper with a short retrospection and lists some ideas for future work.

1.1 The MQTT Publish/Subscribe Protocol

The availability of real-time data from many distributed sources is an important and ever-growing requirement of many enterprise computer networks. Typical examples include data from sensors, such as RFID-readers in supply-chain management or measurements of monitoring systems. Traditional SCADA[1] systems typically use polling to gather data, often based on vendor-specific, proprietary data transport protocols. In a SCADA system, "master stations" are responsible for gathering the data by polling data sources, processing the data, storing it in files or databases, and further distributing it. Especially in large and heterogeneous SCADA systems, the correct acquisition and distribution of data across different transport protocols and host systems poses a difficult problem. Its solution often requires expensive custom applications and is difficult to adapt to changing requirements.

To be able to make more efficient use of such data, close integration of the data sources into the enterprise network is required. The data sources are typically small devices with limited battery power, processing power or available memory and storage. Examples include hand-held devices such as scanners in a warehouse, temperature sensors in a cooling chain, flow meters in oil or gas pipelines, power meters reporting current consumption, weather sensors in remote weather stations, and many more.

[1] Supervisory Control And Data Acquisition

The data flow generated by such data sources differs from that of traditional client-server-applications with a powerful central server sending data to clients. Instead, many distributed data sources send data to a few central servers, where it is gathered, analyzed, and acted upon. Often timeliness of message delivery is also an important requirement, as information is expected to be available in real-time, i.e., within a milliseconds-range. Information from the same source can be relevant to several processing applications; it is thus desirable to have a scalable, generic messaging architecture providing an efficient and simple means to direct data flow from the source to interested parties.

The MQTT architecture [MQTT] provides such a system. MQTT is based on a *Publish/Subscribe* mechanism: rather than sending information directly to each interested party, data sources label each message with a *Topic* and send it to a *Broker*, which is then responsible for its further distribution. Systems interested in that particular topic can subscribe with the broker, and the broker will then forward all information on selected topics to all interested subscribers. This loosely coupled architecture is less complex and more scalable than point-to-point communication, since data sources need not be concerned with the correct distribution of each message. A data source does not need to know who the final recipients of it's messages will be, or even about their existence. This facilitates the deployment of an MQTT-based system in an unmanaged environment, e.g., a warehouse. In such an environment, device failures are expected, and deploying or replacing devices should require as little configuration effort as possible. Nodes can dynamically join and leave the system without requiring extensive re-configuration of the whole system. Also, each message only needs to be sent once by each data source, greatly reducing power consumption of small nodes with limited battery power. The existence of a broker also eliminates the need for complex routing and discovery algorithms executed at the edge devices to discover other devices and route messages to them. All a device needs to know is how to contact the broker. Note that this does not limit the topology of an MQTT system to a simple star, as a broker can act as the subscriber of another (called "bridging").

MQTT was designed to be as light-weight, simple and efficient as possible. While other messaging systems offer similar benefits in terms of reliability, they tend to be more comprehensive, but also more computationally intensive for the peers. The WebSphere MQ system [MQ], for example, provides a feature-rich, robust and reliable messaging platform running on a multitude of architectures. While MQ provides many features MQTT doesn't, such as data transformation between different formats, clustering and load balancing for better performance, message prioritization and many more, MQ requires substantially more powerful hardware and was not designed for low latency and ease of configuration.

1.2 Usage Example

This short chapter shall serve to illustrate the typical usage of MQTT on small, mobile devices in terms of a real-world example.

British car insurance company "Norwich Union" offers their customers the possibility of paying premiums based on their "real" usage of their cars rather than statistical models[norw]. Normally, the expected risk - and thus the premium - of a driver are calculated using statistical data pertinent to the driver's

sex, age, ethnic group, address and the like. With the offered new calculation model, drivers pay premiums based on their actual usage of the car; i.e., how often, where, and when they use the insured car.

To that end, the insurer installs a "black box" into the insuree's car. With the help of a GPS receiver, the box registers the car's usage pattern. Periodically, the gathered data is transmitted over a TCP/IP GPRS connection to the insurer, using the MQTT protocol. Using MQTT allows the insurer to use cheap, small and light hardware and makes it easy to integrate the incoming data flow into the enterprise network. Moreover, MQTT adds only a small overhead to the data transmitted over the slow and expensive GPRS connection.

1.3 Data Exchange between Peers

MQTT is usually run over a TCP connection. The connection is always initiated by the client, the broker never actively opens a connection to a client. TCP is designed to be "reliable" [RFC793]: It ensures that packets lost in the network are re-sent as well as in-order delivery of packets to the application. However, TCP cannot prevent packet loss in the sending or receiving application. If an application or even the sending or receiving host completely crash, unsent packets as well as packets received but not yet delivered to the application will get lost. Also, intermittent loss of connectivity can cause the TCP connection to break and interrupt the data flow on the connection, requiring opening a new connection and causing loss of unsent data in TCP's buffers. This kind of network problem is prevalent especially in wireless networks.

These problems are currently addressed at the application level by MQTT. More specifically, different levels of resiliency are achieved by different operational modes specified by the MQTT protocol (version 3, [MQTT-spec]), called QoS levels. Under some circumstances, loss or multiple delivery of a message might be tolerable. A weather station, for example, might sample the current temperature at specific intervals; since the loss of some of those measurements most likely won't significantly degrade the calculated average temperature or other derivatives, it might be perfectly acceptable. An RFID reader detecting which goods leave a warehouse, on the other hand, must reliably deliver each message once and only once; otherwise, missing or duplicate messages might corrupt the entire inventory. The administrator of the application chooses the QoS level of a node's messages according to the importance of their data and bandwidth and energy consumption constraints. Note that the QoS level is specific to a single message rather than an entire connection.

QoS 0 This message exchange quality level offers at-most-once or "best effort" semantic. The communication consists of a single PUBLISH message sent from the client to the broker. Neither sender nor receiver have a way of detecting the loss of messages.

QoS 1. The reception of a QoS 1 PUBLISH message is acknowledged by a PUBACK message. To identify messages, each message contains a Message-ID header field (which is specific to a connection between a sender and a receiver, not to the message itself). If after some timeout no PUBACK has been received by the sender, the message is re-sent. This scheme offers at-least-once semantic: It is guaranteed that the receiving application will

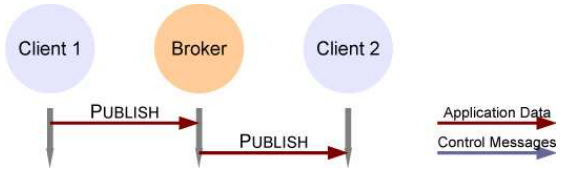

Figure 1.1: MQTT QoS 0 mode

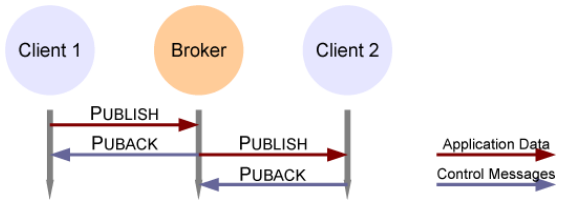

Figure 1.2: MQTT QoS 1 mode

get a message, but the message might also be delivered several times (e.g. due to lost PUBACKs). Until the receiver's acknowledgment is received, the sender stores the message persistently on disk.

QoS 2. This is the highest available service level; it guarantees an exactly-once semantic. This protocol flow is used when the delivery of duplicates is not acceptable. In this scenario, the recipient first receives a PUBLISH message, stores the message persistently, and sends back a PUBREC (*Pub*lish *Rec*eived). When the client receives the PUBREC, it knows that the broker now has a copy of the message, and can thus delete its own copy. This way, a situation where a client stores a message indefinitely although it has already been published by the broker is averted. Upon receiving the PUBREC, the client then sends a second message, a PUBREL (*Pub*lish *Rel*ease). The recipient confirms the PUBREL with a PUBCOMP message (*Pub*lish *Comp*leted). Only after this 2-phase message exchange is completed, the message is processed by the recipient.

The three possible QoS operation modes of the protocol are illustrated in

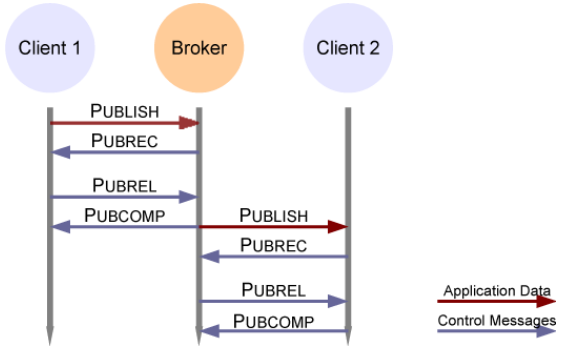

Figure 1.3: MQTT QoS 2 mode

figures 1.1 to 1.3.

It is possible for an administrator to restrict the QoS levels available to a client; a message sent by the publisher with a higher QoS level than available to a subscriber will be re-sent to the subscriber with the highest available QoS to that subscriber.

As can be seen, the additional reliability of higher QoS levels comes at a price:

- It adds additional processing logic to the broker and client applications:
  - Messages need to be stored by the sender until an ACK or PUBREC is received (QoS 1 and 2, respectively);
  - Timers need to be maintained for unacknowledged messages, to detect loss and re-send if necessary
- Higher QoS levels adds additional consumption of bandwidth and transmitting power. The QoS 2 protocol, for instance, uses 3 control messages (PUBREC, PUBREL, PUBCOMP) in addition to each transmitted message containing application data.
  While the control messages are only a few bytes in length and thus do not consume large amounts of bandwidth by themselves, the additional power needed for transmission must not be underestimated. Especially in small sensors using wireless communication, the current consumed by the radio can be many times that consumed by the processor. (Compare, e.g., [SHB+04].) It must be noted that a single sent MQTT message triggers the sending of many other packets: RTS/CTS frames, MAC layer ACKs, and TCP ACKs. In [Per05] it has been shown that even for a simple setup with only one broker, one publisher and one subscriber and a channel error probability of 10%, the amount of bytes exchanged at the MAC layer will more than double for the QoS 2 mode as compared to QoS 0 (sending rate 25 messages per second, message size 200 Bytes).
- Latency is increased. Compared to QoS 0, the latency of a QoS 2 message is increased by at least 1.5 RTTs. This is a problem especially on networks with high latency; GPRS, for instance, has a link round-trip latency of up to 4 seconds [RAI99]. On wireless networks, where data loss often occurs in bursts, the increased number of exchanged messages also increases the risk of such a burst occurring during the message exchange, requiring TCP to re-transmit packets and thus further increasing latency. Moreover, if several peers are in sending range of each other, the additional messages increase the probability of contentions and collisions, adding even more to the latency (and also power consumption). In [Per05], chapter 5.3.4, this sensitivity of wireless networks towards additional messages is examined in detail.
- For QoS 1 and QoS 2 messages, available bandwidth might be underutilized because at any one time, only 1 message can be in-flight. The reason for this limitation is that the MQTT protocol only mandates different message ID numbers for any two different messages, but does not impose any other constraints, such as that message IDs be strictly monotonously increasing. In that, MQTT's message ID fundamentally differs from TCP's

sequence number, and cannot be used to assure in-order delivery to the application. A thread reading incoming messages from the network would need to buffer the messages in the order they are received (TCP does guarantee that this order does not change on the connection) while the message exchange is completed; this is not currently implemented. The current implementation performs the delivery protocol sequentially for each QoS 1 or 2 message. This renders message ID numbers virtually useless for the MQTT network stack (they are, however, used internally by the MQTT application).

The goal of this dissertation is the design and implementation of a solution that alleviates these problems. The design goals are described in the next chapter.

1.4 Goals of ReTCP

The goal of this thesis is to implement a new approach to transmitting messages in a manner resilient to application failures as well as network failures. Resiliency will be added to a connection by lower-level messaging stack modules rather than by the application itself. This frees application developers from concerning themselves with all the tedious details of reliable message delivery. The application will be able to use the simple QoS 0 protocol and still benefit from an exactly-once semantic, if needed. Depending on the requirements, an administrator can choose to use the new ReTCP stack and get QoS 2 semantics, or use the existing network stack for QoS 0 semantics.

The design goals of ReTCP are summarized below.

- ReTCP provides the same level of reliability as the current QoS 2 protocol version; i.e., exactly-once semantic.
- The number of exchanged messages should be smaller than with the current QoS 2 protocol version. As explained above, this will reduce the bandwidth consumption as well as the latency.
- ReTCP allows multiple in-flight messages for increased bandwidth efficiency.
- ReTCP is resilient against network failures and application failures: Even when peers crash, no messages must be lost.
- Flow control: Messages should not be sent to a receiver at a rate faster than it can process the messages, thus filling up its queue of unprocessed messages and possibly causing it to run out of memory.

To this end, data buffers on persistent storage will be used on both the sending and the receiving end of a connection, similar to the buffers used by TCP connections.

Some issues arising with the possibility of failures are not addressed by the proposed solution. In particular, the following problems are not addressed:

- Not processing the same message twice when recovering from an application failure remains the application's responsibility. This task is generally

not easily solved and highly depends on the application at hand.
If the application for instance writes received messages to a database, recovery from a failure is relatively simple and consists of simply checking for the relevant entry in the database. However, if the processing involves e.g. communication with other peers, restoring the exact state before the failure is generally impossible without querying other involved peers. This task cannot be done by the network stack, as it involves many specifics of the application.

- Messages won't be delivered atomically, i.e., the fact that *any* subscriber of a topic has received a message published for that topic does not imply that *all* subscribers of that topic have received it. Such an atomic delivery mechanism that is resilient against site and network failures would require execution of a more complex commit protocol for each message. A formal protocol description of such an atomic, non-blocking commit protocol ("three-phase commit") was first presented in [SS83].
  However, since the MQTT system is designed to be highly dynamic (clients can connect or disconnect from a broker at any time), such behavior would actually be undesired in this context. In fact, a badly behaved peer or even a peer with a unreliable wireless connection would stop other peers from receiving messages in such a setup.

Some features provided by the current design of MQTT QoS levels 1 and 2 will not be provided by ReTCP:

- QoS 1, at-least-once delivery, will no longer be provided. It does not offer any advantages over QoS 2 from an application point of view and requires the same amount of exchanged messages.
- The QoS level will be specific to a connection, not to single messages. An application chooses to use either the existing stack (providing QoS 0), or the new stack (QoS 2) for a connection.

The design of ReTCP is described in the next chapter. An overview of how the implementation was added to the existing MQTT communication stack is then given in chapter 3.

Chapter 2

Design

In this chapter, the MQTT protocol, version 3, shall briefly be explained. For the exact specification, please refer to [MQTT-spec]. Then, the ReTCP design will be illustrated, and the behavior of the MQTT protocol running on a ReTCP stack will be examined.

2.1 MQTT Protocol Overview

2.1.1 Header Format

The generic format of any MQTT message is depicted in figure 2.1.

Fixed Header The MQTT protocol was designed to add as few communication overhead as possible. In its simplest form, the MQTT header is just two bytes in size, as shown in figure 2.1. This *fixed header* is always part of an MQTT message. Depending on the message type, the message will also additionally contain variable header data or a payload.

Message Type The message type is an identifier for the MQTT command to be executed, and is one of the possible types specified in table 2.1.

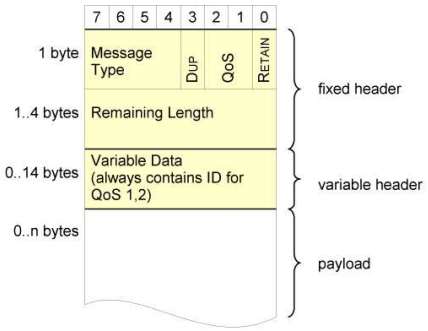

Figure 2.1: MQTT message format

DUP Flag By setting this flag, the sender indicates that the message is a retransmission. The flag is only used for QoS levels 1 and 2, since neither sender nor receiver can detect lost messages in QoS 0.

QoS Level The QoS level of the message, as described in section 1.3.

RETAIN Flag A broker can *retain* a message published to a topic, and immediately send it to new subscribers of that topic. This can be useful when new subscribers need to perform some initialization tasks based on a previously published message, and do not want to wait for a new publication. By setting this flag, a client informs the broker that a message should be retained (replacing any previously retained message).

Remaining Length The number of bytes remaining in the current message, including the length of the variable header data and the payload. For lengths greater than 127 bytes, several bytes are used to encode the length, whereby 7 bits of each byte are used to encode the length, and the 8th bit is used as a "continuation bit". At most 4 length bytes are allowed, thus allowing for a maximal message size of $2^{28} - 1$ bytes (256 MB).

Variable Header Depending on the message type, the variable header contains additional information required for the execution of the command at hand. The variable header data for the different message types is described in more detail in chapters 2.1.2 to 2.1.6.

Messages sent with a QoS of 1 or 2 always include a message ID field. This 2-byte identifier is used to uniquely identify a message in the communication flow. It is inevitable to use such an identifier for the higher QoS levels, since the recipient of a PUBREL, PUBCOMP, or any type of ACK message must know to which original message the received message corresponds to. Message IDs in any communication usually start with 1 and are increased by 1 for each subsequent message. In contrast to TCP sequence numbers, however, this continuous use of IDs is not required by the MQTT protocol specification. This shortcoming in the specification that can lead to under-utilization of the available bandwidth, as described in 1.3.

An ID of 0 is defined as invalid; the message ID size of 2 bytes thus allows for a maximum number of $2^{16} - 1 = 65535$ coexistent messages in any MQTT session. After a message has been PUBACKed or PUBCOMPed (QoS 1 and 2, respectively), it can be deleted and the ID can be reused.

2.1.2 Connection Establishment

As stated earlier, a broker will never connect to a client on it's own; a client willing to join an MQTT system is responsible for connecting to the broker. To this end, the client first opens a TCP connection to the broker, and then starts an MQTT session by sending a CONNECT message.

In addition to the fixed header, this CONNECT message also contains variable header data:

Name	*Value*	*Description*	*Remarks*
CONNECT	1	Connection request from client to broker	
CONNACK	2	Connection acknowledgement	
PUBLISH	3	Message publication	
PUBACK	4	Publication acknowledgement	QoS 1 only
PUBREC	5	Publication receive acknowledgement	QoS 2 only
PUBREL	6	Publication release	QoS 2 only
PUBCOMP	7	Publication completed	QoS 2 only
SUBSCRIBE	8	Client subscription request	
SUBACK	9	Subscription acknowledgement	
UNSUBSCRIBE	10	Client unsubscription request	
UNSUBACK	11	Unsubscription acknowledgement	
PINGREQ	12	Ping message	
PINGRESP	13	Pong message	
DISCONNECT	14	Disconnection request	

Table 2.1: MQTT Protocol message types

Protocol Name (10 bytes) UTF-encoded[1] name of the protocol; fixed to "MQIsdp" in MQTT version 3 (for MQSeries Integrator SCADA Device Protocol).

Protocol Version Number (1 byte) "3" for version 3.

CONNECT Flags (1 byte) Connection options, as described in table 2.2. The flag's meanings are defined as follows:

Clean Start When set, this flag signals to the broker that the client wants to return to a known, "clean" state; any outstanding messages for the client will be deleted, the message id reset to 1, and the client's subscriptions will be reset.

Will Flag By setting this flag, a client can tell the broker to publish a message in its name in case the broker fails to communicate with the client. When no data from the client arrives within the maximal allowed idle time defined by the keep-alive timer field (see below), or an I/O error occurs during communication, the broker assumes the client "died" and will publish the will message on it's behalf; a DISCONNECT from the client does not trigger the sending of the will message.

Will QoS The Quality of Service level the broker should use when publishing the will message.

Will Retain Tells the broker whether or not to retain the will message after it has been published.

Keep Alive Timer (2 bytes) A time period, measured in seconds, that specifies the maximal amount of time between any two messages the client is

[1] By "UTF-encoded", the MQTT specification refers to a modified UTF-8 encoding scheme also used in Java. See http://java.sun.com/javase/6/docs/api/java/io/DataInput.html #modified-utf-8. The UTF encoding includes the string length, therefore no message length field or delimiter character need to be defined.

bit	7	6	5	4	3	2	1	0
flag	unused		Will retain	Will QoS		Will	Clean start	unused

Table 2.2: CONNECT Flags

allowed to let pass. It is the client's responsibility to send a message in each interval. This allows the broker to detect a dropped network connection. In the absence of application data, the underlying TCP connection will not detect this failure[2]. When there are no application data to be sent, the client must send a PINGREQ message.
If no data from a client is received within 1.5 times the specified keep-alive timer interval, the client will be regarded as disconnected by the broker. In case the client had set the will flag on connect, the broker will then publish the client's will message. The client's subscriptions won't be affected by the timeout.
The client can disable this timeout mechanism by setting the timer value to 0.

After this 14-byte variable header, the MQTT specification mandates the following data in the message payload[3]:

Client ID For the broker to be able to maintain a list of subscriptions for each client across many CONNECTs and DISCONNECTs, it must be able to uniquely identify a client when it opens a connection. Therefore, each client has to specify an ID, which has to be unique among all clients connecting to the same broker. If a client with the same ID is already connected, it will be disconnected.
The client ID is UTF-encoded, and must be between 1 and 23 characters in length.

Will Topic (If the will flag is set) When the will message is published, it is published to this topic. UTF-encoded string of variable length.

Will Message (If the will flag is set) The will message that is published. UTF-encoded string of variable length.

Upon reception of the client's CONNECT message, the broker should transmit a CONNACK message. The CONNACK contains a single additional byte as it's variable header, which indicates the success of the connection attempt. A value of 0 means the connection setup completed successfully, the client is now connected and the MQTT session set up. Values 1 to 3 signify a rejected connection attempt due to an illegal protocol version, an illegal client identifier, or an unknown error on the broker side, respectively.

If the connection attempt fails or times out after some client-specified timeout period, the client will close the TCP connection to the broker and retry the whole connection procedure.

[2] Unless TCP's "keep-alive" feature is used. See [RFC1122], section 4.2.3.6.

[3] The MQTT protocol specification explicitly refers to this data as the CONNECTS payload, which is somewhat counterintuitive since this data is intimately related to the connection establishment process and not actual application data.

2.1.3 Subscribing to and Unsubscribing from Topics

Once a client is has successfully established an MQTT session with a broker, it can register it's interest in a particular topic by subscribing to it. It does so by sending a SUBSCRIBE request to the broker. Subscription requests are always sent with QoS 1; hence, the QoS bits in the fixed header are set to 1. Also, the variable header contains a 2-byte message identifier, which will be contained in the SUBACK message to let the client know which message is being ACKed.

The remaining bytes of the message are the payload, comprising pairs of UTF-encoded topic names followed by a byte indicating the desired QoS for the reception of messages posted to that topic.

MQTT's topic space is organized hierarchically, so that clients can vary the degree of particularity of the messages they will get. A topic name is an arbitrary string, however, the characters '/', '#', and '+' have special meanings. Consider the following example: A broker is collecting weather data (temperature, humidity) from weather stations all over Switzerland. The topic space is organized as "weather information"/<region>/<data-topic>, where <data-topic> is one of "temperature" or "humidity".

- A client interested in the current temperature in Zurich will subscribe to "weather information/Zurich/temperature".
- The '+' wildcard is used to match a single level of the hierarchy space. To get temperature information for all regions, a client subscribes to "weather information/+/temperature".
- The '#' wildcard matches a whole subtree of the hierarchy space. To get all available information for all regions, a client subscribes to "weather information/#".

Pre-configuring the broker's topic space is not necessary; when a client subscribes or publishes to a topic that does not exist in the broker's topic space, it will be created. The broker does not impose any limit on the number of topics or the depth of the topic space, either. The only limitation on topics is that a topic name must not be longer than 64kB, and must consist of single-byte characters only.

Unsubscription works similar to subscription. To unsubscribe from a set of topics it has previously subscribed to, a client sends an UNSUBSCRIBE request containing a list of topics. The broker acknowledges the unsubscription with an UNSUBACK message.

2.1.4 Publishing Messages

When connected to a broker, a client can publish messages. A message is always published to a topic, while the precision of the chosen topic can be varied by the use of wildcards, as described above. The topic name is part of the variable header, along with a message ID for QoS 1 and 2 messages.The broker will then publish the message to all clients subscribed to that topic. The payload is entirely application-specific. A client does not need to be subscribed to a topic to be allowed to post messages to this topic.

The actual publication process varies depending on the chosen Quality of Service level, and has already been described in 1.3.

2.1.5 Pings and Pongs

MQTT peers can check other peers by using a ping-pong style query mechanism. A connected peer sends a PINGREQ (Ping request) to another peer, which the other peer answers with a PINGRESP (Ping response). If no PINGRESP is received, the other peer is considered dead. PINGREQ and PINGRESP packets consist only of a fixed header with the according message type value.

As stated before, when specifying a Keep-Alive interval on connect, a client is responsible for sending a packet within each keep-alive period. If no application data is available, it must send a PINGREQ.

2.1.6 Disconnection

A client disconnects from the broker by sending a DISCONNECT message, which only contains of a fixed header of the according message type. The broker does not acknowledge the DISCONNECT.

Any current subscriptions of the disconnecting client are not affected by the disconnection. Also, messages sent with a QoS greater than 0 and with the Retain flag set, will be stored and delivered to the client when it reconnects.

2.2 ReTCP from a Bird's Eyes View

Before diving into the design details of ReTCP, the design goals and their expected benefits are shortly restated here. Since ReTCP is implemented in the context of the MQTT network stack, the behavior in an MQTT system is of primary interest; note, however, that other applications can use ReTCP as well.

This section also lists the assumptions on which the design is based.

2.2.1 Assumptions

Network The design of ReTCP assumes ReTCP is run over a TCP connection. It explicitly relies on some of TCP's features:

- Error detection and correction.
  It is assumed that TCP detects and handles errors introduced at lower network layers, such as loss or flipping of bits.

- In-order flow of bytes.
  As will be explained later, ReTCP offers a packet-oriented connection; the application writes data in the form of packets to the ReTCP layer. In contrast, TCP is stream-oriented; it regards application data simply as a stream of bytes. While ReTCP uses sequence numbers to keep packets in order, it relies on TCP to do so for the byte order within a packet.

- Flow Control.
  TCP uses sophisticated algorithms to adapt its sending speed to the current state of the network and the receiving application. Most notably, it implements congestion-control and -avoidance algorithms to react to packet loss due to congestion while using available bandwidth most efficiently, and receiver-controlled flow control in the form of advertised window sizes, to stop a fast sender from overwhelming a slow receiver.

	# msgs	*bytes sent*	*round trips*
Existing Stack, QoS 0	1	$n + 2$	1/2
Existing Stack, QoS 1	2	$n + 4$, 4	1/2
Existing Stack, QoS 2	4	$n + 4$, 4, 4, 4	3/2
ReTCP Stack, QoS 0	2	$n + 5$, 3	1/2

Table 2.3: Comparison of QoS 0 over ReTCP vs. higher QoS levels over TCP, for message with n application bytes. Reading example: A QoS 2 message of n bytes causes 4 messages to be sent. The first message is n bytes in size plus additional 4 bytes for the MQTT QoS 2 header. Each additional message is 4 bytes in size. The delivery latency is 3/2 round trip times.

> These algorithms come in many different flavors and have been fine-tuned in many years. Rather than re-inventing the wheel, ReTCP tries to make optimal use of TCP's flow control.

Application Failures It is assumed that the program execution at either communication peer can be interrupted or stopped due to a power loss, hardware failure, user interrupt or similar condition. We have to assume that such an interrupt can happen at any time in the program flow, i.e., we cannot guarantee the atomic execution of any piece of code.

However, we assume that byzantine failures as defined in [LSP82] do not occur. As long as a peer is sending data to other peers or writing data to its disk, this data is assumed to be correct and consistent.

2.2.2 Expected Behavior

To summarize 1.4, ReTCP is designed to:

- Provide the same level of resiliency towards network failures as the current MQTT QoS 2 protocol mode;
- Be robust against application failures as described above;
- Reduce the number of exchanged messages;
- Decrease the latency for message delivery;
- Reduce the overhead of protocol header bytes vs. application bytes;
- Increase throughput by improving parallelization.

The interesting performance indicators will be throughput, protocol overhead, and latency. A simple analysis of the operation and header formats of ReTCP according to the next section allows an estimate of the behavior of ReTCP. The number of total bytes sent, messages exchanged, and required round-trip times until the message is available to the receiving application are summarized in table 2.3.

Interestingly, as the comparison shows, we can expect a QoS 0 message flow over ReTCP to perform almost identically to a QoS 1 flow over TCP; the number of exchanged messages, the protocol overhead in bytes, and the latency are

identical! Merely the distribution of the protocol overhead is slightly different, because the MQTT ACK uses a 4 byte header, while ReTCP only adds 3 bytes.

Compared to the QoS 2 MQTT mode, we see a dramatic decrease in latency. This will be especially welcome in applications where timeliness is key. Also, the overhead in bytes is halved, and only 2 instead of 4 messages are exchanged. On busy wireless networks, this seemingly small difference can bring a large performance boost, since fewer and shorter messages reduce collision probability and therefore can reduce the amount of sent bytes at the MAC layer to a much greater degree. This is especially true for a publisher that sends small messages at a large rate, possibly only a single reading of an integer value. Even on networks with a small collision risk, sending fewer messages means less energy consumption and longer battery life.

The effects of using ReTCP on throughput are somewhat more difficult to estimate. Fewer exchanged messages and the parallelization of the connection increase the throughput, while the added processing time of ReTCP could have a detrimental effect. The actual performance gain in terms of throughput will largely depend on message size and exchange rate and the efficiency of the implementation, and can only be assessed by a test setup using the actual implementation.

2.3 Design of the ReTCP Protocol

Developed over 40 years ago, the TCP protocol today is more widely used than ever, and has proven its flexibility and reliability over and over again. While adding some features to TCP, as discussed in section 1.4, the design of ReTCP is similar to TCP and incorporates some of TCP's proven mechanisms to reach it's goal of being resilient towards network and application failures. In particular, the following elements are used in ReTCP's design:

- Reliable Delivery.
  All received application data is acknowledged to the sender. Only when a packet has been acknowledged by the receiver, the sender will delete it from its sending buffer.

- In-Order Delivery.
  While TCP ensures in-order delivery for packets received over the same TCP connection, it cannot do so for packets received over several connections. To ensure in-order delivery for packets received over the same ReTCP-connection, a 32-bit sequence number is used on the ReTCP-layer as well.

- Sliding Window Protocol.
  Like TCP, both peers use a sliding window to keep track of sent and received packets, so that multiple packets can be in-flight at any time. In contrast to a TCP buffer, however, this send and receive buffers are persistent; whenever a packet is added to or deleted from them, the buffer contents are immediately written to disk so that they survive even machine crashes.
  The peers can also advertise a window size, to limit the size that the receive buffer needs in memory and on disk.

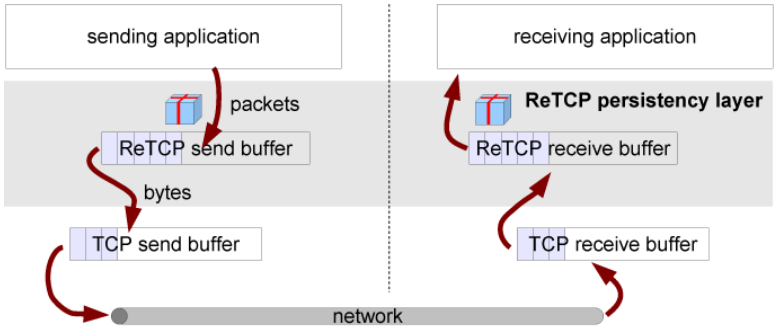

Figure 2.2: ReTCP-based network stack

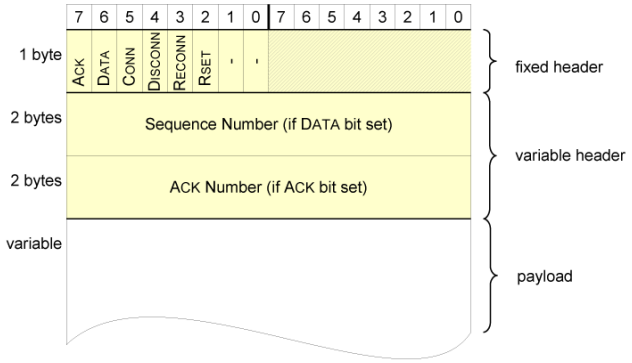

Figure 2.3: ReTCP Header Design

The interaction between the TCP-layer, the ReTCP-layer, and the application is shown in figure 2.2. Like every network layer, ReTCP encapsulates packets received from higher layers in it's own headers. The header format is shown in figure 2.3 (note that the figure displays one half-word per line, not one byte like the MQTT header illustration). Like the MQTT header, the ReTCP is designed to be small, so as to add as few overhead to the communication as possible. Again, the header comprises a fixed header and a variable header.

Fixed Header The fixed ReTCP header contains only a single field: A byte containing flags. The connection flags have the following meaning:

ACK The ReTCP packet contains an Acknowledgment. Usually, this means that an application data packet in the packet flow is being acknowledged; however, this flag is also used to acknowledge connection and disconnection.

DATA If set, indicates that the packet contains application data.

CONN By setting this flag, the sender indicates it wants to set up a ReTCP session.

DISCONN By setting this flag, the sender indicates it wants to terminate a ReTCP session.

RECONN When a previous ReTCP session was uncleanly disconnected due to a network or application failure, the state at the time of failure will be restored on reconnect when this flag is set.

RSET When the previous state cannot be restored, the reconnection attempt will be terminated by setting this flag.

The flags and their usage are explained in more detail in the following sections 2.3.2 and 2.3.3.

Variable Header The variable header contains data relevant for connection setup and the synchronization of the states of both peers. Its content varies depending on the message type, and will be explained in detail in sections 2.3.2 and 2.3.3. Except for the packets required for connection setup, it can contain only two additional fields:

Sequence Number (2 bytes) The sequence number is present in packets that contain application data (i.e., have the DATA bit set). It is used to identify duplicate or missing packets in the stream of incoming packets, just as the TCP sequence number. The TCP sequence number cannot be used to that purpose, since it will not be continuous for packets belonging to the same ReTCP connection but sent over different TCP connections (and also because upper network layers should not rely on such internals of the lower levels; in fact, the TCP sequence number will usually not even correspond to a single ReTCP packet!).

ACK Number (2 bytes) Packets with the ACK bit set will include this header field. The value of the field is a cumulative acknowledgment, letting the sender know that the receiver has received all packets up to the one with this sequence number.

2.3.1 Persisted Connection Data

To be able to keep state of a ReTCP session among several TCP connections and possibly even in the face of intermediate machine failures, it is necessary to store some state information of ongoing ReTCP sessions. This state needs to be stored on persistent storage, where it can survive crashes and reboots. This state information needs to be updated frequently and kept in a consistent state with the other peer's persisted state, requiring some additional control data exchange between the two peers.

The most important information that gets persisted are the send- and receive-buffers, as shown in figure 2.6. The persisted information is described below.

- The send buffer, including all unacknowledged packets
- The SND.UNA pointer, set to the first not yet acknowledged sent packet sequence number
- The receive buffer, including all unconsumed packets

Note that the SND.NXT pointer needs not be saved; it can be reconstructed from the send buffer state when it is reloaded. Similarly, the RCV.NXT pointer of the receive buffer indicating which packets have already been acknowledged needs not be held on persistent storage; it can be rebuilt the first time an ACK is sent, as it simply describes up to which sequence number all packets have been received. If the receive buffer is empty, it will be set to the sequence number of the first packet received, plus one. The SND.UNA pointer is needed in case the send buffer is empty; in that case, it corresponds to the sequence number of the next packet that will be sent, and SND.NXT will be initialized to SND.UNA.

Apart from the buffers and the packets inside them, some "accounting information" for each connection also needs to be stored:

- The maximum buffer size for both send and receive buffer
- Connection information that matches the persisted state to a certain connection, where a connection is identified by a connection ID and a peer name, as described in 2.3.2
- The time the state was last modified, for garbage collection

2.3.2 Connection Setup and Termination

Session Handshake A ReTCP connection between two peers (not necessarily an MQTT client and broker, as the range of application of ReTCP is not limited to this scenario) is established as follows. The client wishing to communicate over the ReTCP layer first opens a TCP connection to the server[4]. After the TCP connection is set up, i.e. the TCP 3-way handshake has succeeded, a ReTCP session is established. To this end, another handshake is performed, this time on the ReTCP layer. The client initiates this ReTCP session handshake by sending a CONNECT packet, which is described in figure 2.4. The connect contains the following variable header data:

Connection ID Within MQTT, any client is only allowed to have at most one connection to a broker at any time. Other applications, of course, do not have such a limitation. In that case, the server listening for client connections must be able to match an incoming ReTCP connection to an existing ReTCP session or create a new ReTCP session if none exists. To enable the server to distinguish different ReTCP connections, each connection is assigned a connection id.
This ID is chosen by the client, and the server does not care about how it's chosen. It is the client's responsibility to ensure that the (client ID, connection ID) tuple is unique.

Session Validity In case the ReTCP connection to a client breaks (due to a failure of the server, the client, or the network), the server will keep all relevant data of this connection on persistent storage, so that the connection can be re-established. However, it might be desirable to limit the lifetime of the persisted data, so that data of clients that break and never

[4] To a well-known port number where the server listens to ReTCP connections. Usually a server will also listen on other ports for "TCP-only" connections, for message exchanges that do not need the resiliency offered by ReTCP.

reconnect does not fill up the server's disk over time. With the session validity field, a client can tell the server how long to store data pertinent to this connection. This is only a hint for the server; the server can still choose to ignore this value and use a different one, see below. If set to 0, the client indicates to the server that it wants to use the maximal allowed time span allowed by the server. Otherwise, the time span is measured in seconds, allowing a maximal validity of $2^{32} - 1$ seconds, well over a century, which should be sufficient for all practical purposes.

Peer Name Length The length, in bytes, of the client's name. This should equal the remaining amount of bytes in the message.

Peer Name The client's name. Again, the server does not care how the client chooses this name, but it must be unique among all clients connecting to this server. In the context of MQTT, this requirement is pretty simple to fulfill, since there already is such a client name, the MQTT client ID. However, several alternatives would work equally well:

- Clients could choose a unique hardware-related value as their name, e.g., the MAC-address of their NIC.
- The IP/port combination, if it is assumed to remain static.
- The hostname/port combination, if it is assumed the the hostname will always resolve to the same physical machine.
- If no such value is available to the clients, they could randomly choose an ID. If the ID space is chosen large enough for the number of clients in a system, and the generation of the ID is reasonably "random", the probability of a collision can be made arbitrarily small.
- If none of the above approaches work, a small extension to the ReTCP protocol could be used to solve the problem: When a client connects to a server for the very first time, it asks the server to assign a name to it by setting a flag in the CONNECT message. The server, in it's reply, includes a unique name that this client is henceforth allowed to use.

Upon reception of a CONNECT request, the server sends a CONNACK. The CONNACK has the same structure as the CONNECT packet but will have the CONN and ACK flags set, and the RECONN flag if this was set in the request. Unless the RSET flag is also set, the ReTCP session handshake was successful, and the client can now start sending data. The connection establishment between an MQTT client and a broker, including messages on the TCP, ReTCP, and MQTT layer, is shown in figure 2.5.

The session validity field in the server's response indicates to the client how long the server is willing to keep data relevant to this session around. The server should use a value less or equal to that requested by the client in the CONNECT request.

Re-establishing ReTCP Sessions When performing the handshake described above, both peers need to know whether they are establishing a new ReTCP session or resuming a previously uncleanly disconnected session. In

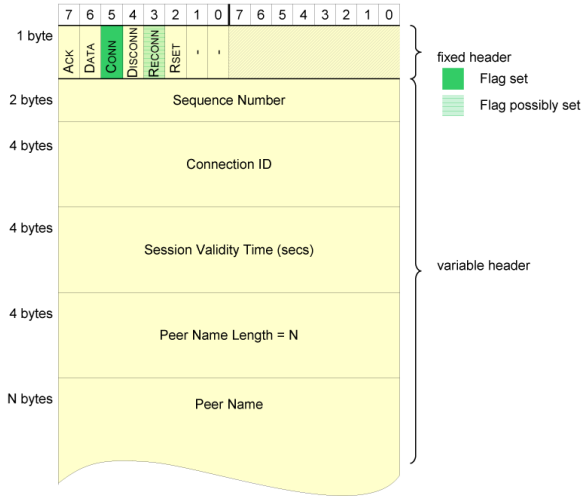

Figure 2.4: ReTCP CONNECT Packet

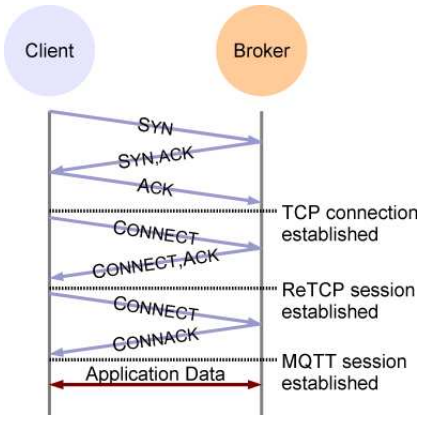

Figure 2.5: MQTT connection establishment over ReTCP stack

the latter case, both peers will have state information for this session stored persistently.

Therefore, before connecting to the server, the client checks whether state exists for the server address and port it wants to connect to. If such persisted state exists, and the client decides it want to resume the according session, it sets the RECONN flag in the CONNECT request. Normally, the previous state is then reloaded by both peers during the handshake.

It is possible that a client crashes or gets disconnected during termination of a ReTCP session. in that case, it is possible that the client failed to delete the state for a session, while the server deletes it. If a client tries to restore a session for which the server does not have data, it should set the RSET flag in its response. When the client receives a CONNACK with the RSET flag set, it should delete the persistent state for this session, and reattempt the handshake with the RECONN flag not set. It should be noted that this situation merely indicates that the ReTCP session termination was interrupted; a server must never delete state for a session when there are packets not yet delivered to the application, so no application packets are lost even when a handshake attempt fails due to a RSET.

Even when the state of a previous session does exist, a client can choose to ignore it by not setting the RECONN flag during the handshake. When a server receives a connection request with an unset RECONN flag, it must discard any previously persisted state information, even when there are packets not yet delivered to the application.

Terminating ReTCP Sessions A session is disconnected by either of the peers sending a DISCONNECT message. Unlike TCP, a ReTCP session cannot be half-open; disconnecting a session always closes data streams in both incoming and outgoing direction. After sending the disconnection command, a peer must no longer send packets to the other peer. Upon reception of a DISCONNECT, a peer should no longer accept new packets from the application. A DISCONNECT message must be acknowledged with a control message with both the DISCONN and the ACK bit set.

A disconnection request or acknowledgment must not be sent while there are still unsent packets in the send buffer. Also, the ReTCP layer should stop accepting new packets from the application layer once the DISCONNECT or disconnection acknowledgment, respectively, have been sent. Both peers must, however, deliver any remaining packets in the receive buffer to the application. Upon sending or reception, respectively, of the disconnection acknowledgment, both peers should delete any persistently stored data of that session.

2.3.3 Data Flow

Sliding Windows Once the peers have set up a ReTCP session, they are ready to send and receive data. The data flow much resembles that of the underlying TCP: both peers use a buffer for packets received from the lower (TCP) layer (the receive buffer), and a buffer for outgoing data received from the upper application layer (the send buffer). The data in the buffers corresponds to the contents of a "sliding window" moving over the stream of packets, whereby each packet is assigned a sequence number to identify its position in the stream.

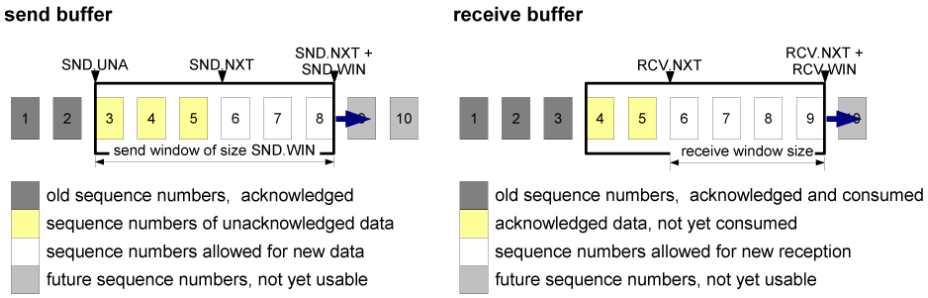

Figure 2.6: Sliding Window Protocol

The sliding window protocol is illustrated in figure 2.6. New packets are added to the right-hand side of the buffers. In the illustration, the next sent packet would be added at position 6. The send window position is always such that all packets to the left of it have been acknowledged. A cumulative acknowledgment scheme is used, i.e., an ACK message with an ACK header field value of n means the receiver has received all packets up to and including n[5]. The send window is moved forward to the right by every acknowledgment received that acknowledges one or more previously unacknowledged, consecutive packets starting from the left side of the window. The receive window position is always such that all packets left to it have already been acknowledged to the sender. The size of the receive window depends on the amount of available buffer space and the amount of data that is buffered and acknowledged but not yet consumed by the application.

In contrast to TCP, the buffers states are persistently stored and need to be updated on certain events. Every time the send or receive window advances, packets to the left of it can be deleted from persistent storage, since they are delivered to the application (in the case of the receive window) or acknowledged by the other peer (send window). Also, packets added to the send buffer will immediately be persisted, so that the sending application is assured that the packet will not be lost. Note, however, that there is no need to immediately persist the receive window when new packets arrive over the network, since these packets are still persisted in the sending peer's send window. The packets must be persisted before the according acknowledgment is sent.

The meaning of "delivered to the application" also differs from that of TCP in the context of ReTCP. For TCP, delivering a packet simply means that the application has read it from the buffer. Since ReTCP aims at dealing with application failures as well as network failures, this is not good enough. An application might read a packet and then crash before processing it. ReTCP guarantees that even in that case, no data loss will occur. Therefore, a confirmation from the application is needed that it is done processing the packet. This confirmation can be either an explicit notification of the ReTCP layer by executing a callback function in the stack whenever a packet has been processed, or it can be achieved implicitly by having the ReTCP stack execute an API function of the application that only returns after the packet is successfully processed.

In TCP, each peer must advertise the currently available space in its receiving

[5]"packet n" standing short for "the packet with sequence number n"

window. The sender is not allowed to use a larger send window size at any time. This receiver-controlled flow-control is to avert situations where a fast sender overwhelms a slower receiver by sending more data than the latter can handle. ReTCP does not need such window advertisements, because it can rely on the flow control of the underlying TCP. The receiving ReTCP can simply use a buffer size limit suitable for the present hardware and application requirements, and stop reading data from the TCP layer when this limit is reached. When the application using ReTCP cannot drain the buffer as fast as the sender is transmitting new data, the ReTCP buffer will fill up, ReTCP will stop reading data from the TCP layer, causing the TCP receive buffer to fill. At this point, TCP's flow control will kick in and stop the sender from transmitting too much data. A receiver could even shrink it's receive window during a ReTCP session without affecting the sender (a behavior which is theoretically possible for TCPs as well, but strongly discouraged by [RFC793]).

Packet Retransmission Again, because TCP is used as the basis for our persistency layer, it is not necessary to re-invent the wheel for the detection and correction of packet loss. Along with flow control, TCP already perform this tedious task, and performs it well. However, in some situations, additional steps have to be performed for in-order and reliable delivery.

1. TCP can lose unsent data if a peer gets completely disconnected. When a peer cannot be reached anymore (be it due to the network failing, the machine crashing, or the operating system crashing), the other peer's TCP will eventually time out, tear down the connection, and discard any unsent data in the send buffer[6]. Further attempts to send data by the application will fail.
   In ReTCP, data loss is prevented in this situation, because unsent data is persistently stored. When the ReTCP session is re-established, however, TCP's guarantees for reliable, in-order delivery no longer apply. ReTCP has to ensure this for any packets in it's send buffer.

2. When the network connection and the receiving host are OK, but the receiving application (including the ReTCP layer) hangs, TCP will happily deliver any new data to the receiving peer as long as the TCP receive buffer does not run full. As an example, think of a Java VM that hangs, while the underlying operating system is running normally. The sending application can still write data to the network, but the receiving application will never read it. In the absence of new application data eventually filling up the receiver's TCP buffer, this situation might persist indefinitely. Using a normal TCP connection, the sending application has no way of telling whether the receiving application has actually read a sent packet from the TCP layer.

Problem 1 can be solved relatively easily by the use of sequence numbers. They allow the receiver to add the packets to the ReTCP buffer in the correct order even across multiple TCP connections, and allow the sender to detect lost packets after session re-establishment by the absence of their acknowledgments.

[6] see 4.2.3.5, "TCP Connection Failures", in [RFC1122]

Note that 1 will never cause missing acknowledgments as long as the underlying TCP is connected.

To speed up the synchronizing process of the two peers in case of a session re-establishment, a "fast retransmit" algorithm is used. When a session is re-established, and the first data packet sent in the reestablished session has sequence number n, and the next received ACK has an ACK header field value of $m < n$, then all packets between m and n, exclusive, are immediately resent.

Problem 2 is requires a somewhat more involved solution. ReTCP solves it by using retransmission timers for sent packets the same way TCP does. If no acknowledgment is received for a packet by the time of the retransmission timeout (RTO), the packet is re-sent. Since this retransmission timeout period must take into account the time required for the Transmission on the TCP layer and the time required by ReTCP for adding the packet to its receive buffer and persisting it, a fixed RTO value cannot be used. As stated before, a ReTCP packet can have any size between 1 byte and 256 MB, and depending on the application, packet sizes during a ReTCP session might have a large variance. How the retransmission timeout is calculated depending on packet size is described in the next section.

Using RTO-based retransmissions on top of a carrier that already provides reliable delivery is not as straightforward as it might seem at first glance. The first pitfall an implementation needs to avoid is known as the "TCP meltdown" effect [HOI+05]. Note that ReTCPs timeout period will usually be longer than TCP's, since it also includes the processing time on the ReTCP layer. Because the TCP layer adds a level of indirection to ReTCP's timeout calculation, this is not necessarily true at all times. The RTO values of both TCP and ReTCP are updated based on received acknowledgments. Since acknowledgments on the TCP layer can be more frequent than those on the ReTCP layer, TCP will adapt its RTO more quickly. Sudden changes in the RTO by TCP can thus lead to an inverted situation where TCP's timeout is larger than ReTCPs.

Such a situation can lead to a meltdown, dramatically decreasing or even completely stalling a connection. The meltdown effect occurs when two transmission protocol layers that both use RTO-based resends are stacked upon each other, and when packet loss due to congestion is present. The lower layer will detect the congestion more quickly and adapt its RTO, possibly to a value larger than the RTO of the upper layer. The upper layer will then start retransmitting packets at a faster rate than the lower level, adding to the congestion and therefore further slowing down the connection, which in turn triggers even more unnecessary resends on the upper layer. This vicious circle is well known in TCP-over-TCP applications such as VPNs where TCP traffic is tunneled through a TCP-based SSH connection (see, for example, [Titz01]). Observe that the meltdown can only happen when there is unsent data in TCP's send buffer.

Another potential pitfall is that the absence of an acknowledgment within the RTO does not necessarily mean that there is something wrong with the other peer or the TCP connection. An acknowledgment can also simply be delayed because it is queued behind a large packet at the other peer. While ReTCP is busy writing the large packet to the network, the ACK can obviously not be sent.

Based on above considerations, the following retransmission algorithm is executed when a timeout expires. Let RTO_{TCP} := TCP's current RTO, and

RTO_{ReTCP} := ReTCP's current RTO.

1. When a thread is currently reading a packet from the network, wait until this thread has read the entire packet. If the next incoming packet is the missing ACK, return.
   To avoid waiting forever when the other peer has crashed or the network fails, only wait while data is being read. If no data is received within RTO_{TCP}, proceed to the next step.

2. When the outgoing TCP buffer is empty, resend the packet. Backoff ReTCP's RTO (see 2.3.4), and return.

3. If $RTO_{ReTCP} < RTO_{TCP}$, backoff the ReTCP RTO, but do not resend the packet, since this situation could lead to a meltdown. Re-schedule the retransmission timer with the new RTO value.
   A value $MAXRTO_{ReTCP}$ is used as the upper bound for this doubling operation. When $2 * RTO_{ReTCP} > MAXRTO_{ReTCP}$, do not re-schedule a retransmission; instead, proceed to step 4.

4. If $MAXRTO_{ReTCP}$ is exceeded, it is likely that we cannot send any more data, either because the remote peer has crashed, or because of a network failure. Therefore, pause all operations writing to the network, and wait for the TCP send buffer to drain. When the send buffer has drained, it means data can still be sent; in this case, the timed-out packet is now resent.
   Again, to avoid waiting forever, when the TCP send buffer size does not change within RTO_{TCP} the remote peer or network is assumed dead. In that case, the ReTCP session is terminated. Unsent packets remain of course persisted, so the session can be resumed later.

In a situation where the receiving application hangs but manages to recover after some time, its ReTCP layer will restart reading data from the TCP buffer and send acknowledgments. If previously read packets have not been lost, the re-sent packets will then be duplicates, which the ReTCP layer of the receiver will notice due to the sequence numbers, and not deliver them twice to the application. In the (perhaps more likely) event that the receiving application never recovers, ReTCP will give up the retransmission of a packet after a fixed amount of tries (currently 10, but configurable). When a packet is not acknowledged after this number of retries, the receiver is considered "dead", and the ReTCP connection closed. If this event is logged, it can serve as hint to the systems administrator that something with the receiving application is wrong and needs fixing; a clear advantage over the "TCP-only" connection, where this event might go unnoticed, as described in 2.

Operation To enable ordering and retransmission as described above, the ReTCP header for data packets contains ACK data as depicted in figure 2.3.

To reduce the number of ReTCP packets, delayed acknowledgments can be used on the ReTCP level. It must be considered that using delayed ReTCP ACKs can harm TCP's own delayed ACK implementation ([RFC1122]) when the sum of packet processing time and delay of ReTCP is larger than TCP's delay.

Also, TCP implementations should implement the Nagle algorithm ([RFC896, RFC1122]). The Nagle algorithm avoids sending many small application writes as single TCP packets. It states that small amounts of data are not sent immediately when there are still outstanding TCP ACKs from previously sent data. Instead, data is buffered and sent only when all outstanding data has been ACKed, or a full-sized TCP segment can be sent. What this means for the ReTCP layer is that subsequent ReTCP packets are likely to be sent as a single segment on the TCP layer. Therefore, a ReTCP ACK followed by a ReTCP data packet are likely to be sent in a single TCP segment. Compared to a single ReTCP packet containing a delayed ReTCP ACK and the application data, only 3 bytes for the ReTCP header are wasted. So using delayed ACKs on the ReTCP level might save 3 sent bytes per received data packet, but perhaps more likely, it prohibits TCP from sending the ReTCP ACK together with the delayed TCP ACK, thus wasting 40 bytes for the TCP and IP headers.

Given that for QoS 0, a client publishing a message will not get a reply from the broker, and that most clients will have a mean message inter-arrival time greater than a few hundred milliseconds typically used for TCP's delayed-ACK timeout, at least in the context of MQTT disabling the use of delayed ReTCP acknowledgments and enabling nagling is clearly preferable.

Implementation Requirements To assure that recovery from an application failure is possible when the respective communication end-point restarts, a few design principles need to be followed in the implementation of the protocol:

- Received packets are only ACKed *after* they are added to the persistent inbound buffer.
  ACKing packets before adding them to the buffer could lead to lost packets, if the end-point fails after sending the ACK but before adding the packet to the buffer.

- Received packets can only be deleted from the inbound buffer *after* complete processing by the application.
  Normally, a packet is removed from the network buffers as soon as it has been handed to the application for further processing. However, since the application can potentially crash during processing, packets need to remain in the inbound buffer until the application signals to the persistency layer that a particular packet has been fully processed (e.g. its data stored to disk or sent to other nodes) and is no longer needed.

- The application layer needs to be notified when sent packets are persisted.
  The persistency layer guarantees that no packets are lost once they are received from the application layer and added to the outgoing buffer. The application in general will want to know from when on this is the case. This can be achieved by either having the application use a synchronous method call of the persistency layer that returns only once the packet is persisted, or by explicitly notifying the application that a packet has been received and persisted by a control message or callback method.

2.3.4 RTO Calculation

When retransmission timers are used to resend failed packets, it is key that these timers are dependable. Too small RTO values will cause unnecessary resends, which wastes sending power and may add contention to the network. Too large RTO values, on the other hand, mean that a sending application will be unnecessarily slowed down when packet loss occurs.

[RFC2988] describes how the RTO is calculated for TCP. The algorithm is listed in appendix A.1. The parameters the algorithm uses are those suggested by [JK88]. The basic idea behind the algorithm is to base the current timeout on sampled round-trip time measurements from the past, and also take into consideration the measured variance[7] of the RTTs. A variable $SRTT$ contains the *smoothed* average of all past RTT measurements. The average is calculated according to an exponential weighting of the measurements; each new measurement is added to the $SRTT$ with twice the weight of the previous measurement. This way, newer measurements are more relevant for the current value, which allows the RTO to adapt quickly to changing network conditions, but sudden jumps in the RTO due to few outliers in the measured RTTs are avoided. $RTTVAR$, the round-trip time variance, is updated in a similar fashion.

Acknowledgments for re-sent segments are ambiguous, since it is unclear whether they are a response to the original transmission or a retransmission. Therefore they are excluded[8] from the RTO calculation [KP87].

Unfortunately, TCP's RTO algorithm cannot be used by ReTCP without modification, because TCP is stream-based, while ReTCP is packet-based. Remember that a packet can have any size up to 256 MB. If ReTCP were to use TCP's RTO algorithm, and an application would send many small packets, each taking e.g. 2 seconds to transmit, then the RTO would quickly converge to a value close to 2 seconds. If the application then sent a 256 MB packet, it would almost immediately time out. On the other hand, the RTO would be greatly skewed by large packets, yielding a too large value for small packets. Using TCP's RTO directly clearly leads to inadequate RTO values. An algorithm also taking into account the packet size is needed.

The segment sizes TCP sends can vary, too; still, TCP does not include the segment size in its RTO calculation[9]. One might ask why the TCP retransmission algorithm works well in spite of this fact. The answer is that segment sizes in TCP are limited. By default, this MSS limit is 536 bytes; during connection setup, a TCP may indicate that a larger MSS can be used for that connection [RFC879, RFC793]. For an Ethernet network, for example, the maximal IP datagram size is 1500 bytes, yielding a TCP MSS size of 1460 bytes. The time needed to send a segment depends on the network bandwidth and the segment size. Even for very small bandwidths, the additional time required to write

[7] The term "variance" is actually misused by [RFC2988]. $RTTVAR$ measures the mean deviation rather than the variance or the standard deviation. The mean deviation is simpler to calculate and an approximation of the standard deviation. This paper sticks to the convention of the RFC, and the mean deviation is referred to as "variance".

[8] Unless TCP's timestamp option is used. It allows the sender to place a timestamp in the TCP header, which the receiver will echo in it's Ack, allowing the disambiguation of the Acks and a more frequent update of the RTO, which can improve performance.

[9] TCP does, however, take into account that the RTT may change due to an increase of the send window during the slow-start phase. This is why the RTTVAR is multiplied by 4 in the algorithm. The assumption is that doubling the window size, in the worst case, also doubles the RTT.

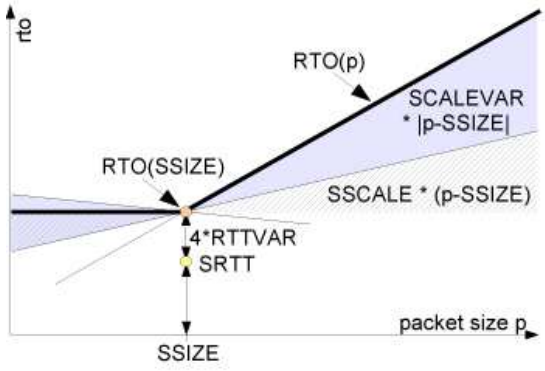

Figure 2.7: ReTCP RTO Calculation

a segment of size MSS compared to writing the smallest possible segment (21 bytes, 20 for the TCP header and 1 byte of application data) is negligible for all practical purposes. The same is obviously not true for ReTCP, where packet sizes can vary within many more orders of magnitude.

For ReTCP, an algorithm that computes the RTO based on packet size is required.

The proposed algorithm is based on the following assumption: the round-trip time of a packet increases linearly with its size, with a fixed initial latency. More precisely, $RTT = 2l + \frac{p}{R} + D * x + y$, where l is the link latency, p is the packet size, R is the link bandwidth, x the processing delay of the ReTCP layer per data unit, and y a fixed latency of the ReTCP layer. Thus, the RTT takes the form $RTT = a * p + b$, where a and b are unknown. While this assumption clearly holds true for the network, it has to be verified for an actual ReTCP implementation.

In the proposed algorithm, the smoothed round-trip time is a point in the two-dimensional plane defined by the two axes packet size and RTT. Therefore, a smoothed size value $SSIZE$ is introduced, that is updated the same way as $SRTT$. Whenever the RTO is updated with a new measurement, the point $(SSIZE, SRRT)$ moves along the line defined by $a * p + b$. This allows us to compute the *smoothed scaling factor* $SSCALE = a$, which is the slope of this line. Just as for the $SRTT$, the measured variance for $SSCALE$ is also maintained as $SCALEVAR$ and used when computing the RTO.

The RTO with this algorithm is no longer a fixed scalar value, but a function of the packet size:

$$RTO(p) = RTO(SSIZE) + (p - SSIZE) * SSCALE + |p - SSIZE| * SCALEVAR$$

The ReTCP RTO calculation is summarized in figure 2.7.

Because the 2-dimensional ReTCP RTO algorithm has more unknowns than the scalar TCP RTO algorithm, the "uncertainty" of a calculated $RTO(p)$ value is larger. Note that the algorithm is designed in a conservative way where the variance of both the SRTT and the scaling factor add to the calculated RTO. Especially when the algorithm is not in a “steady state”, i.e., only few measurements have been made or the parameters a and b fluctuate, the variance

of the calculated RTO can grow very large. Therefore, it makes sense to place additional upper bounds on the RTO calculation:

- $RTO(p)$ should not be larger than $RTO(SRTT)$ if $p \leq SRTT$.
- $RTO(n * p)$ should not be larger than $n * RTO(p)$
- additionally, an absolute upper bound based on the characteristics of the network and the receiving application may be placed on the RTO.

The proposed algorithm is listed in appendix 4. Note that for constant packet sizes, it equals the TCP RTO algorithm.

Simulating the RTO Calculation To verify that the algorithm indeed produces good results, i.e. the calculated RTO converges towards the actual round-trip times, the behavior of the algorithm has been simulated for several scenarios.

- The first scenario aims to find out how fast the computed RTO converts towards a constant RTT. The RTT for a single MSS is simulated to be 2 seconds.
- Scenario 2 considers a slow change of the characteristics of the underlying network. Round-trip times slowly and steadily increase from 2 to 3 seconds.
- The last scenario investigates how the algorithm reacts to a sudden increase of the RTT from 2 to 3 seconds.

The round-trip times used in all 3 scenarios have a deviation σ of 10% of the current RTT value.

Keep in mind that the assumption of the algorithm is that RTT times depend on packet sizes. Hence, the RTT values defined above are *minimal RTTs* of a packet of minimal size. Additionally, a linear delay between 0 and 1 seconds depending on the packet size is added to the minimal RTT.

The simulation calculates both TCP's and ReTCP's RTOs. It is assumed that a ReTCP packet of size $n * MSS$ causes n TCP segments to be sent, which generate exactly n ACKs so that each TCP Ack can be used to update the TCP RTO. (I.e., no packet loss or duplication is assumed). Furthermore, it is assumed that the nth TCP ACK and the ACK for the ReTCP packet arrive at the same time (i.e., delayed TCP ACKs are used, and the TCP ack piggybacks on the ReTCP ACK). The simulation results are shown in figures 2.8 to 2.10. The x-axis shows the number of received TCP-ACKs, and the y-axis the current retransmission timeout for both TCP and ReTCP.

The first figure shows that both RTOs converge towards the RTT and reach a steady state after some 15 or 20 received acknowledgments. In the third simulation, the result of the exponential RTO backoff is clearly visible, and both RCP and ReTCP require around 20 to 30 ACKs before the RTO is adapted to the new RTT. Perhaps most interesting are the results of the second scenario. As can be seen, ReTCP reacts more heavily to RTT changes than TCP does. The ReTCP graph shows some pronounced spikes and tends to be more conservative than TCPs. This is not surprising bearing in mind that the ReTCP algorithm has a greater degree of uncertainty, and the variance in both dimensions the

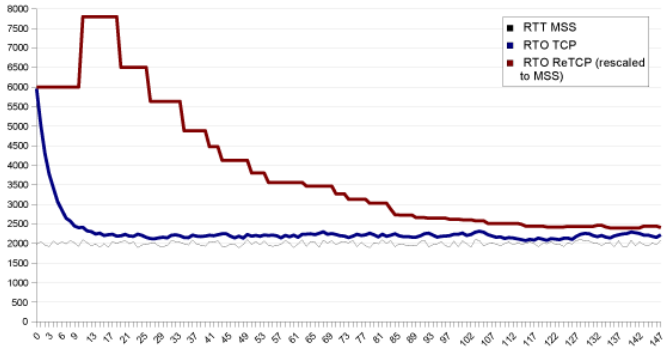

Figure 2.8: Simulated RTO convergence towards constant MSS RTT

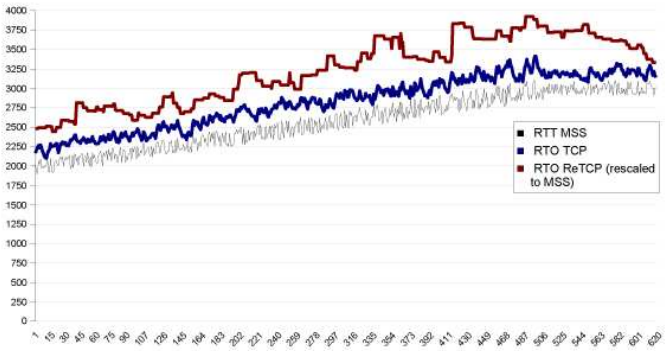

Figure 2.9: Simulated RTO calculation for steady changes of RTT

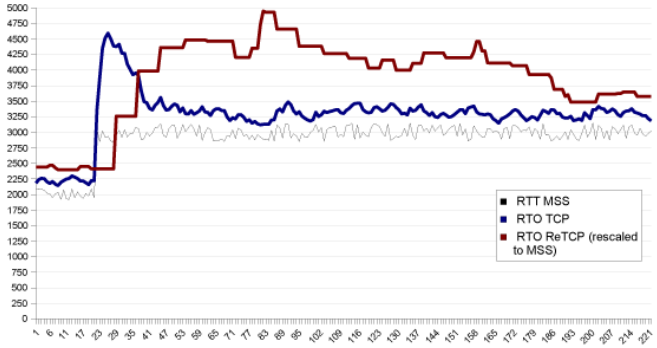

Figure 2.10: Simulated RTO calculation for sudden change in RTT

algorithm operates in add to the resulting RTO. It is important to note, however, that in all 3 scenarios the ReTCP RTO converges to a value close to the RTT after about the same amount of received ACKs as TCP's RTO.

Chapter 3

Implementation

The first section of this chapter describes the design of the existing MQTT network stack. The next section then shows how ReTCP was implemented within this architecture.

3.1 The MQTT Communication Stack

3.1.1 Stack Architecture

The MQTT protocol it is implemented in a variety of languages for different platforms. The implementation discussed here was written as an addition to the Java-based MQTT communication system. The Java-based communication stack consists of several modules stacked on top of each other, each performing its specific task in the processing of the incoming and outgoing data streams; hence the name "stack", which does not refer to the stack of layers of the TCP/IP network layer model. The entire MQTT communication stack is part of the application layer in the TCP/IP model.

An important design goal of the MQTT stack is flexibility. Therefore, the stack can be dynamically composed of an arbitrary amount of modules, for which the only requirement is adherence to an `IProtocolHandler` interface. This modular design was directly inspired by the design of the x-kernel Protocol Framework [xkernel]. By virtue of this design, the stack functionality can be extended with, e.g., new protocols without the need for recompilation of the client or broker libraries. The modular design also facilitates code re-use, because the same module can be used as part of different stacks. To replace a TCP- by a UDP-based stack, for instance, it suffices to replace the lowest-level module in the stack, the Net module. The overall architecture of an MQTT stack is depicted in figure 3.1. A short description of each component is below.

While TCP provides a stream-oriented API, data within the MQTT stack flows in packets. A packet essentially consists of a payload of application data, and a list of headers that the modules modify while the packet is passed along the chain of modules. Each protocol module has the possibility to add a header when a packet is moved downwards and remove it when a packet is moved upwards, but is not required to do so.

The individual stack modules are loosely coupled. The packets flow asynchronously between the modules, coordinated by a singleton dispatcher.

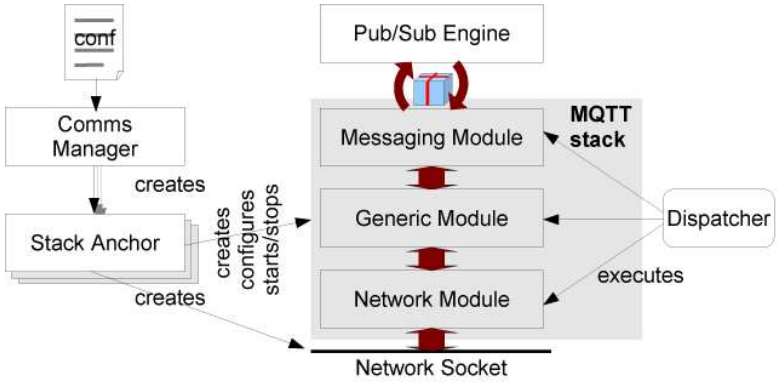

Figure 3.1: MQTT communication stack

Dispatcher The dispatcher is responsible for passing packets between the different stack modules. Rather than directly passing packets to other modules, a module wishing to send a packet to its upper or lower neighbor hands the packet to the dispatcher, together with the target. The dispatcher makes sure every module is only processing at most one packet at any point in time, freeing the module developers of the tedious task of thread synchronization of the module's data structures.
The dispatcher also has flow-control mechanisms. When a module indicates it has trouble keeping up with the processing of packets, the dispatcher will stop delivering packets to this module, until the module informs the dispatcher that the overload situation has cleared and it is ready again.
Besides packets flowing trough the stack, the dispatcher also handles timers. Modules that want to be informed after a given amount of time has passed can register a event, and will be called back after the specified time by the dispatcher, so they can process this event. Timers can be scheduled once-only or periodic.
There is exactly one dispatcher per JVM that is used by all stack instances concurrently.

Protocol Anchor The protocol anchor manages stack instances of a certain type. The life-cycle management of the stack is performed by the anchor; it instantiates, starts, and stops the stacks. On the broker side, protocol anchors are *listeners* waiting for client connections; on the client side, *activators* instantiate the stack when the client opens a connection. Section 3.1.3 explains the stack instantiation process in more detail.

Comms Manager The comms manager is responsible for the entire communication system of the client or broker. It maintains a set of protocol anchors (listeners and activators) according to the system configuration.

Pub/Sub Engine The pub/sub engine is the part of the MQTT application that communicates with the stack. As its name implies, its responsible for publishing messages and handling subscription requests.

Messaging Module This is always the topmost module of the stack. It implements the MQTT protocol. Here is where, e.g., a PUBREC message is generated when a PUBLISH is received with QoS 2. The messaging module will pass the PUBLISH message to the Pub/Sub engine when the 2-phase message exchange is complete.

Net Module Is the lowest stack module and interfaces to the network. It converts packets it receives to a stream of bytes that are written to the network socket, and creates packets from data read from the incoming network byte-stream (called "deframing"). To that end, a small header indicating the packet length is used. Network I/O is done either using a thread or using Java's NIO.

Generic Modules Between the Messaging and the Net module, an arbitrary number of generic modules can be plugged into the stack. The can provide functions such as data transformations, encryption, segmentation / reassembly, and much more.

3.1.2 Anatomy of a Module

Besides methods for life cycle management (`init`, `start`, `stop`), a module offers various handler methods to handle sends, receives, and timeouts. The modules do not call these methods directly on each other; it is the dispatcher's task to execute the protocol stack by calling the handler methods of the stack modules. A module provides the following handlers:

`handleSend` gets called by the dispatcher as a packet traverses the stack in the sending direction. Once the module has finished the processing of the packet, it returns control over the packet back to the dispatcher by executing a callback method. The dispatcher in turn calls `handleSend` on the next-lower module. Once the packet has reached the lowest module, it is written to the connection socket.

`handleReceive` is used in a similar manner as `handleSend`, but for received packets traveling in receiving direction through the stack.

`handleTimeout` is called by the dispatcher when a timer that the module has registered expires. To let the module know which timer expired (as a module can register any number of timers), the timer is identified by an ID that was assigned when it was registered. Additionally, a module can register an arbitrary object on timer creation, which will be passed back to the module in the `handleTimeout` handler.

`shutdownSend` is executed when the sending direction of the stack is being closed.

`shutdownReceive` is executed when the receiving direction of the stack is being closed.

3.1.3 Stack Instantiation

How and when the stack of a peer is created is somewhat different for brokers and clients. In both cases, the stack configuration is dynamic, and each available

stack type is described by a list of module names and optionally some parameters for the modules. The comms manager translates the module names into Java class names, creates a protocol anchor for each type, and passes the list of class names and parameters to it.

Broker The broker configuration defines a set of listeners and also specifies what port they should listen on. On broker startup, a `ServerSocket` is created for each listener and waits for client connections on the specified port. When a client connects, a the listener creates a new stack, and passes the connection socket to it.

Client On the client side, a stack is instantiated each time the client decides to connect to a broker. Which type of stack gets instantiated depends on the connection scheme the client uses to specify the broker address. For TCP, the scheme is `tcp://<broker address>:<broker port>`. For ReTCP, a new scheme, `retcp://`, has been added to the list of available stack types.

Once a stack anchor (listener or activator) decided to instantiate a stack, the process is the same for client and broker:

- The anchor creates all module objects with the given list of class names.
- Each module is given a reference to its upper (receiving direction) and lower (sending direction) neighbor.
- The stack is initialized by passing module-specific parameters to each module.
- The stack is started by calling a `start` method on each module. The required steps to start a module depend on the module; a thread-based Net-module will, e.g., start a thread reading from the connection socket when its `start` method gets called.

3.1.4 Stack Termination

A stack instance is terminated by shutting down the data streams in both sending and receiving direction. The sending direction is shut down by a "shutdownSend" command traveling downwards in the stack; i.e., the dispatcher calls `shutdownSend` in one module after the other in sending direction, starting from the module that initiated the shutdown. Once the "shutdownSend" has reached the lowest module, it is converted into a "shutdownReceive" and its direction reverted; it travels back up the stack, until it is back at the module that initiated the shutdown. Similarly, a "shutdownReceive" initiated by a module is sent upwards, converted into a "shutdownSend" at the topmost module, and travels downwards until it reaches the initiator again.

In normal operation, a "shutdownSend" is initiated by the topmost module when the MQTT application wants to disconnect from another MQTT host and closes the stack. The "shutdownReceive" is normally initiated by the lowest module, when no more data from the network is available. For instance, consider an MQTT client that wants to disconnect from a broker (TCP connection). It

sends a DISCONNECT MQTT message to the broker and then closes its sending direction. When the "shutdownSend" has received the Net module, the Net module closes the outgoing TCP connection. Once the incoming TCP connection has been closed by the broker, the Net module sends a "shutdownReceive" up the stack.

When an error occurs in the stack, the module where the error occurred will shut down the stack. In that case, it sends both a "shutdownSend" and a "shutdownReceive".

Whichever module initiated the shutdown waits for the shutdown messages to return and then informs the stack anchor that the stack has shut down. This allows the anchor to update its data structures and remove the now defunct stack.

3.2 The ReTCP Network Stack for MQTT

3.2.1 Module Design

The modular composition of the stack poses some problems when implementing the ReTCP protocol described in chapter 2. The ReTCP stack needs to communicate with the application on top of the stack as well as with the underlying TCP buffers.

Synchronous communication with the application is required because packets must be persisted as soon as they are handed to the stack from the application, and the stack must not delete packets from persistent storage before they have been processed by the application. The loose, asynchronous coupling of the modules through the dispatcher makes this difficult unless the ReTCP module is the top-most module in the stack.

On the other hand, ReTCP also needs access to the TCP layer to avoid the TCP-meltdown problem. To avoid unnecessary resends, a packet must not be resent at the ReTCP level when there is still unacknowledged data in the TCP buffer.

There are several ways of solving the problem that the resilient stack needs to communicate with the application as well as the TCP layer. A discussion of the different approaches follows below.

1. A monolithic stack.
   In this solution, the stack design is greatly simplified by using a single module doing all the work, from communicating with the application to checking TCP buffers.

2. Additional synchronization messages within the stack.
   The ReTCP functionality could be split up into two modules, one at the top and one at the bottom of the stack. The top-level module is responsible for the communication with the application, the persistence of the messages, and the resilient delivery to the other network peer. As an interface to the TCP layer, a second module at the bottom of the stack is used. The two modules communicate by passing additional messages up and down the stack.

3. Emulating MQTT QoS 2 within the stack.
   This is a variant of solution 2; additional messages within the stack are

used for synchronization. Unlike 2, however, all ReTCP functionality is implemented in a single module at the bottom of the stack. Synchronization with the application is reached by using the MQTT QoS 2 protocol. This is possible because the QoS 2 protocol mode offers exactly-once-semantic.
When using the ReTCP stack, an application is required to use the QoS 2 protocol. The ReTCP module emulates the remote peer and performs the 2-phase message exchange for every application message. During the 2-phase message exchange, the message gets persisted at the ReTCP level, and then sent to the remote peer with the protocol described in chapter 2. Similarly, received messages are delivered to the application by the 2-phase QoS 2 protocol.

From a design point of view, 2 is the cleanest way to implement ReTCP within the existing architecture: It retains the modular design of the stack, allowing arbitrary other modules to be plugged into the ReTCP stack. Because synchronization messages between the 2 ReTCP modules are only necessary when the upper ReTCP module detects a time-out of a message, the performance overhead added by the synchronization messages should be small.

Solution 3 seems like a very natural choice, as it requires few modifications because the already implemented QoS 2 message exchange protocol is used. However, it introduces several problems. Its design requires the lowest-level Net-module, which should only be responsible for providing the transport layer, to know about application-level protocols. This makes the implementation much less generic. It also requires that the QoS 2 protocol mode be part of future MQTT versions, eliminating one of the benefits of ReTCP. Moreover, the amount of messages flowing through the stack is increased by a factor of 4 as compared to solutions 1 and 2. This could have a very adverse effect on performance, although the grade of the performance degradation is hard to estimate without measurements taken with a real implementation.

As a proof-of-concept that the design of ReTCP works in practice and can be used to offer resiliency towards application and network failures to MQTT in a much more beneficial way than the current QoS 2 protocol, it has been decided to implement the solution described in 1.

The implemented ReTCP stack hence comprises a single module responsible for everything from reading packets from the network, to managing persistence, to communication with the application. Several modules that operate independently in the normal MQTT stack are part of the monolithic ReTCP stack. Figure 3.2 outlines the various elements of the ReTCP stack module and their interaction.

StackState The current state of the ReTCP stack is managed by a `StackState` object. The `StackState` manages the send- and receive-buffers along with accounting information such as SND.UNA, RCV.NXT, and buffer sizes. The state is updated each time a packet is received from the application by the `handleSend` method, and each time a packet is received from the network by the `handleReceive` method.

The `StackState` object is responsible for persistence. On each update, it writes state information as described in 2.3.1 to the disk, so that the current

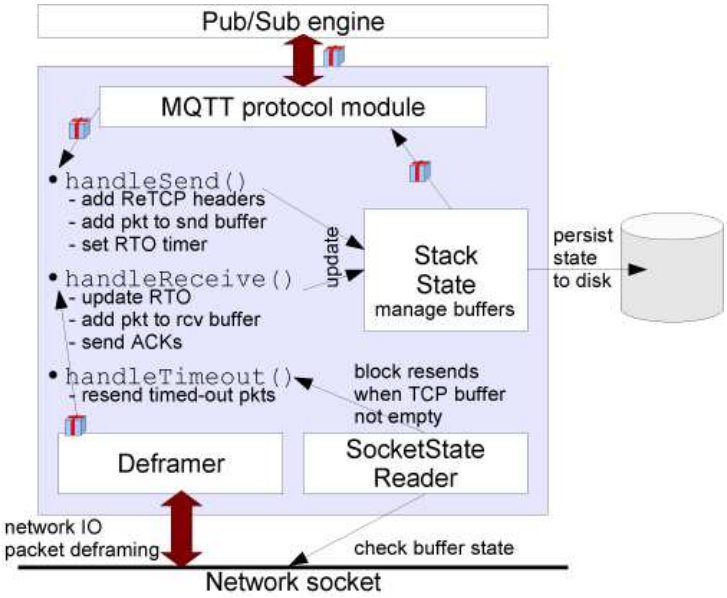

Figure 3.2: ReTCP stack module

state can be reloaded after an application failure. After a crash, the state before the crash is reestablished by calling the `StackState`'s `resurrect` method.

SocketStateReader To execute the retransmission algorithm presented in section 2.3.3, the ReTCP layer needs information about the current state of the TCP layer. Namely, it needs the current TCP RTO, send buffer, and receive buffer states.

On machines running Linux, this information is found in the `/proc/net/tcp` and `/proc/net/tcp6` pseudo-files. For each TCP-connection, these files contain a line with the local and remote port number and address (depending on whether an IPv4 or IPv6 connection is used, the connection is listed in the `tcp` or `tcp6` file), and the required timer and buffer information, along with many other values that are mainly of interest for kernel debugging. The `SocketStateReader` object provides these values to the ReTCP layer by reading them from the relevant `tcp[6]` file. Obviously, this class only can be used on Linux based machines, where the `/proc` file system is available. For other operating systems, this class needs to be replaced.

3.2.2 Persistent Data Store

Persistency Framework To persist data, the `StackState` object uses IBM's `ObjectManager` Java library. The Object Manager allows Java programs to store and update objects under the scope of a transaction, i.e., with guaranteed ACID properties:

- Changes of a transaction are committed atomically;

- Changes of a transaction are isolated from other transactions in other threads;
- Changes of a committed transaction are durable.

Java objects managed by the Object Manager are persistent and survive JVM or even machine restarts[1]. To guarantee atomicity of transactions, a write-ahead log is used, as in many database systems. The Object Manager is optimized to be as fast and lightweight as possible. In can handle write rates of up to around 30 MiB per second and is capable of managing data structures that do not even fit into the available memory.

To manage an object with the Object Manager, it is first allocated to an `ObjectStore`. The store corresponds to a file on disk where the serialized object will be stored. Any updates and changes to the object are then made on an in-memory copy of the object. When the transaction commits, the changes are first written to the write-ahead log, and then the object data on disk are updated.

When an object is allocated to a store, a *token* for this object is created and returned to the caller. The token is a small indirect reference to the object that later allows retrieval of the object from the store. When an object containing tokens is added to a store, the serialization process stops whenever a token is reached, as the token identifies another object that has already been added to a (possibly different) store. Objects can also be added to a store with a name assigned to them, and can later be retrieved by name rather than from a token. This is necessary as a starting point for the retrieval process.

When an Object Manager is instantiated, it performs some expensive initialization tasks, such as allocating space for log files and object store files on disk. This initialization process is relatively slow. Therefore, a singleton `ObjectManager` instance is used by all instances of ReTCP stacks a host is currently using. The assumption is that the write rate of 30 MiB/s is sufficient to persist the combined data flow of all ReTCP stacks.

ReTCP State Data Structure The ReTCP layer on each host needs to store persistent information for each connection, whereby a connection is identified by a peer name and a connection identifier. For this mapping, ReTCP uses a *state table*. In this table, the (peer name, connection ID) tuple is matched to a persisted `StackState` object. Figure 3.3 illustrates this mapping.

When a new stack is created, the data in the state table is updated. To be able to find this table after a restart, it is added to the object store under a well-known name. The next section describes how this process works.

3.2.3 Instantiation of the ReTCP Stacks

In MQTT, the broker never initiates a connection on its own; it is always the client's responsibility to open a connection to the broker, be it in normal operation, after an unexpected disconnect of an MQTT session, or when recovering

[1] The Object Manager also offers other degrees of "persistency". A developer can choose to only persist object on clean shutdown, or to not persist anything at all across restarts. In the context of ReTCP, both these strategies are of no interest.

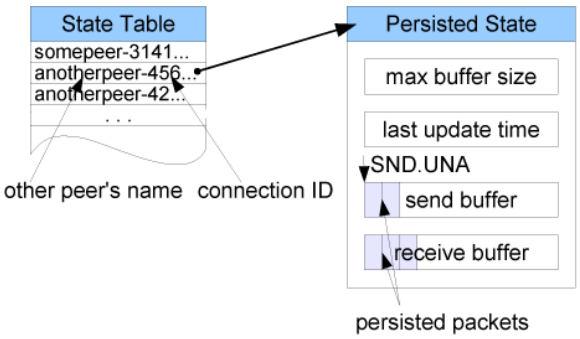

Figure 3.3: ReTCP state table

from a crash. ReTCP does not change this; a broker will not try to reestablish disconnected ReTCP sessions, but simply wait for the client to reconnect.

The connection process is as follows.

Client Side

1. The application at the client side decides it want to connect to a broker. It creates a new ReTCP stack by using the ReTCP stack activator.
2. During instantiation of the stack, it is checked whether an Object Manager instance has already been created. If so, this instance will be used. Otherwise, an Object Manager instance is created. The state table is retrieved from the object store, using the well-known name assigned to it.
3. A lookup for previous connections to the broker is performed. The key used for the lookup is the broker address.
4. If the lookup is successful, a token pointing to the persisted stack state is retrieved from the table. The state is recovered from persistent storage. The connection ID used during the previous connection is part of the persisted stack state.
   If the lookup is unsuccessful, a new stack state is created. The new stack state will contain empty send and receive buffers, and a newly chosen connection ID.
5. The client opens a TCP connection to the broker, and then sends an ReTCP connection request. The connection request includes the client's peer name and the connection ID either recovered from persistent state or newly chosen in 4. If the lookup in 4. was successful, the client sets the RECONNECT bit of the connection request.
6. The client waits for the connection acknowledgment from the broker. When it times out waiting for the acknowledgment, or the acknowledgment has the RSET bit set, the connection setup process is aborted, and an exception is raised. In the latter case, the persisted stack state is also wiped out, because the broker signaled it does not allow the client to reestablish the old ReTCP session.
7. The ReTCP session has successfully been (re)created.

Broker Side

1. The ReTCP listener receives the TCP connection request generated by the client in 5, and creates a new ReTCP stack reading to and writing from this connection socket.

2. During instantiation of the stack, it is checked whether an Object Manager instance has already been created. If so, this instance will be used. Otherwise, an Object Manager instance is created. The state table is retrieved from the object store, using the well-known name assigned to it.

3. The client's ReTCP connection request is read from the network. A lookup for previous connections to the broker is performed. The key used for the lookup are the peer name and connection ID specified in the request.

4. If the RECONNECT bit of the connection request is set, but the lookup in 3. was unsuccessful, a connection acknowledgment with the RSET bit set is sent to the client, and the connection setup process is aborted.
   If the lookup in 3 was successful, but the RECONNECT bit was not set by the client, a new stack state is created, with the connection ID set by the client. The persisted old state is replaced by the newly created one.
   Otherwise, the state is recovered from persistent storage, and the connection acknowledgment is sent to the client.

5. The ReTCP session has successfully been (re)created.

Chapter 4

Testing the Implementation

In chapter 2, we the goals and expectations for the newly implemented ReTCP MQTT stack were set, and its specifications defined. The ReTCP stack needs to be tested for conformance with the specification and fulfillment of the requirements. Also, several tests measuring the performance of the new stack have been designed. This chapter describes which experiments have been executed, and their results.

4.1 Unit and Integration Tests

Several tests were set up to assert that the different parts of the stack as well as the entire stack module do what they are supposed to do. Unit test were used to assert the functionality of the main classes used in the stack. Most importantly, the `StackState` class was unit tested to make sure that all data is persisted correctly and can be fully restored after a crash. Tests to assert that the send- and receive-window data structures are updated correctly when a new packet is sent or received were designed as well.

In a stack module composed of many sub-modules each responsible for one specific task, perhaps more importantly than unit-testing the sub-modules is testing the entire module. To this end, several integration tests have been set up.

Tests should not rely on the correctness of any of the code that they are testing. Therefore, rather than instantiating two ReTCP stacks and letting them communicate with each other in different scenarios, all tests directly create a socket connected to a broker running ReTCP, and write hand-crafted headers according to the specifications in chapter 2 to the socket. They then read the broker ReTCP stack's response from the socket and verify it against the specification. The implementation has passed the following tests[1]:

- The client can connect, and the broker's response is as defined in 2.3.2. The connection id in the broker's response is the one chosen by the client.
- The client can terminate a ReTCP session by sending a DISCONNECT, and gets a disconnection Acknowledgment from the broker.

[1] "Broker" stands short for the tested ReTCP stack run by the broker. "Client" stands for the integration test.

- An aborted ReTCP session can be resumed by the client. The next received packet does not have a larger sequence number than any persisted packet.
- Even when the state of an aborted ReTCP session exists, the broker does not resume the session when the RECONNECT bit in the connection header is not set.
- Trying to resume a non-existing or terminated ReTCP session will result in a connection RSET by the broker.
- Trying to resume an aborted session after the specified validity period will result in a connection RSET.
- The ACK-algorithm is executed correctly by the broker. Sending a sequence of packets n trough $n+m$ results in an ACK for $n+m$. Sending packet $n+m+2$ thereafter still results in an ACK for $n+m$. Sending the missing packet $n+m+1$ results in ACK $n+m+2$.

4.2 Performance Tests

When evaluating the performance of a system, many metrics can be measured, and often they can be measured in different ways. Also, performance often depends on many external factors. This is especially true for network protocols and their implementation, which depend on network bandwidth and -delay, error rates, congestion, and many more factors. Before doing any experiments, a testing methodology has to be chosen, external factors influencing the performance have to be defined, and a set of metrics needs to be selected.

4.2.1 Test Environment

Metrics When the new ReTCP protocol was designed in chapter 2, one of the goals was improved performance in terms of throughput and message delivery latency. It will be tested how well the new ReTCP implementation performs in terms of those metrics, compared to the existing stack running the QoS 2 MQTT protocol mode.

Methodology - Simulation vs. Emulation In an early stage of this thesis, the idea was to test the ReTCP protocol using the OMNeT++ network simulator [omnet]. OMNeT++ is a powerful discrete event simulation framework for communication networks. Simulation models are implemented by C++-modules, that can be hierarchically assembled into larger components and connected with each other trough a high-level model description language. Many parameters of the module connections, such as propagation delay, bandwidth, and bit error rate, can be defined. OMNeT++ is a generic framework, but many modules implementing TCP, IP, various MAC-layers, and many other protocols are available. An extensive performance simulation of MQTT QoS levels 0 to 3 is provided in [Per05].

One advantage that simulation provides is complete control over the testing environment. As every networking layer is completely simulated by OMNeT++,

Scenario	*Ethernet*	*Wifi-5*	*Wifi-10*
Bandwidth (Mbit/s)	100	54	54
Delay (ms)	0.3	1.5	1.5
Drop %	0	5	10
Drop Correlation	n/a	.8	.8

Table 4.1: Factors for different Test Scenarios

it is possible, for instance, to simulate how transmissions on a wireless MAC layer of several hosts affect each other. This is important for finding out how long transmissions get delayed due to collisions, and how many bytes are sent at the MAC layer. The latter, as mentioned earlier, is an indicator for power consumption. Measuring these metrics on real hard- and software under repeatable conditions is a very difficult task.

On the other hand, the drawback of using a simulation is that it tests a protocol rather than its actual implementation. In the case of ReTCP, every time a packet is received or sent, it is persisted to disk which involves several disk writes. Since disk writes are slow, and there are potentially hundreds of them per second, simulating the effect that persisting packets has on the overall performance is impossible through simulation.

Therefore, the original idea of using the OMNeT++ simulation framework was abandoned in favor of a network emulator. An emulator allows to imitate certain properties of a testing environment without the need of special testing hardware.

For the following measurements of throughput and latency, National Institute for Standards and Technology's NIST Net network emulator has been used [NISTNet]. NIST Net is a Linux kernel module that acts as a special software router for IP packets, and allows to emulate various different network conditions. Among other factors, it allows to define:

- The delay of a connection, and the delay's standard deviation
- Available bandwidth of a connection
- The amount of dropped datagrams, and the drop correlation

The authors of NIST Net refer to it as a "network-in-a-box", a single connection hop that models the behavior of an entire network. Network effects are selectively applied to datagrams passing through the box depending on source address, destination address, and optionally source and destination port and protocol type. These rules can be added and changed at run time, without having to restart the module. According to the developers, NIST Net can easily handle thousands of rules and high speed connections even on throw-away hardware.

Emulated Scenarios Finally, the external factors emulated by NIST Net have been defined for 3 particular test scenarios, as listed in table 4.1.

As implied by their monickers, the Ethernet scenario aims at modeling a typical wired Ethernet connection with a very low latency and no dropped packets, whereas the two Wifi scenarios model a 802.11g connection with moderate (5%)

and high (10%) loss rates. For both wireless connection, the drops are highly correlated with a linear correlation factor of 0.8. The correlation factor indicates the conditioned probability of the channel of staying in the "bad" (dropping packets) or "good" state for the next packet. The high correlation factor is used to model the typical error burstyness of wireless channels.

Unfortunately, since NIST Net operates at the IP level, it does not allow for the modeling of the behavior of the Wifi MAC level. Hence, how the amount of bytes sent at the MAC level is affected by the different MQTT network stacks cannot be measured with this test setup. (The delay introduced by network collisions, additional RTS/CTS frames etc at the Wifi MAC layer can to some degree be accurately modeled with the relatively high latency.) Fortunately, though, as stated earlier the message exchange pattern and byte overhead of the ReTCP protocol is identical to the one of a QoS 1 message exchange. The ratio of sent application bytes to bytes sent at the Wifi MAC layer for various scenarios has been analyzed in [Per05], and because of their similarity, the analysis applies to the ReTCP protocol as well.

Hardware The results of every performance test are of course greatly impacted by the environment the test runs in. All tests described in the rest of this chapter were performed on a test machine, *vaticano*, with the following specifications:

- Processor type and -speed: Intel Pentium III, 1133 MHz
- 512 MB of RAM
- Operating System: Linux 2.6.8
- 20 GiB, 5400 rpm IDE hard disk drive, read speed ~15 MB/sec, write speed ~10 MB/sec on an ext3 filesystem

4.2.2 Processing Overhead

Round-Trip Times and Packet Sizes When defining the RTO algorithm in 2.3.4, the assumption was that round trip times increase linearly with the packet size, with a fixed initial latency. This is obviously true for a network link of constant bandwidth. However, whether this also holds true for an application processing the packets depends on the application and has to be verified. If the assumption were wrong, the RTO algorithm would yield inaccurate RTT estimations. Inaccurate RTT estimations can have a very detrimental effect on performance, thus the RTO algorithm would have to be redesigned if the assumption were false.

A first, simple performance test therefore measures the implementation's processing overhead for different packet sizes. Since we are only interested in the processing time in the stack, and not by any delay introduced by the network, the measurements are executed over a link with virtually no delay. A broker and a publisher are both run on vaticano and connect via the local loopback interface.

With this setup, the round-trip times for different packet sizes up to 16 MB are measured. The publisher sends single messages to the broker, and measures the elapsed time until an ACK for that message is received. The result is shown

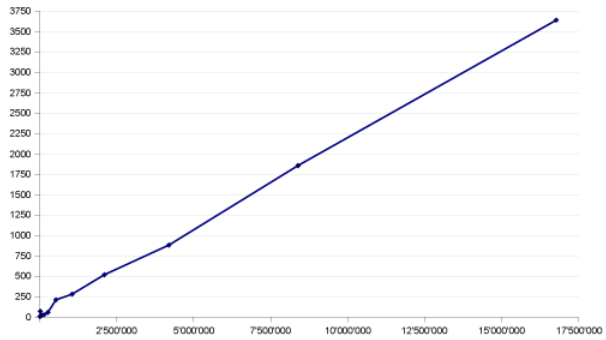

Figure 4.1: Round-trip times (ms) for packet sizes between 1 byte and 16 MB

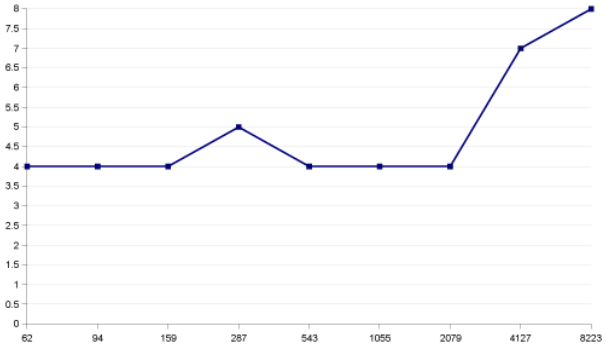

Figure 4.2: Round-trip times (ms) for packet sizes up to 10 kB

in figure 4.1. Packet sizes above 16 MB have not been tested, because they are completely irrelevant for the use cases of MQTT.

The figure shows that the RTT indeed rises almost perfectly linearly with the packet size, with an initial latency of around 4 ms for packets up to 10 kB. Figure 4.2 shows an enlargement of the RTTs for packet sizes up to 10 kB. Here we see that there is only a fixed delay independent of the packet size, and that the packet size only starts to influence the processing times when it is above 2 kB. For this range of packet sizes, the function is not linear. However, for the RTT calculation, this is irrelevant because the RTT algorithm dictates a minimal RTT of 1 second, and the nearly constant delay of 4 ms up to packet sizes above 2 kB is well below that limit.

Note that for most MQTT applications, the range of tested packet sizes is well beyond any message size that will ever be produced. The point of the test was to prove that the initial assumption holds true even for very unlikely scenarios. However, in all further tests, only packet sizes[2] between 10 bytes and 15 kilobytes are used to provide a more informative result for a typical MQTT usage scenario.

Detailed Analysis of Packet Processing The next test was a more detailed investigation of the amount of time the different steps involved in the processing

[2]Application data, without MQTT or ReTCP headers

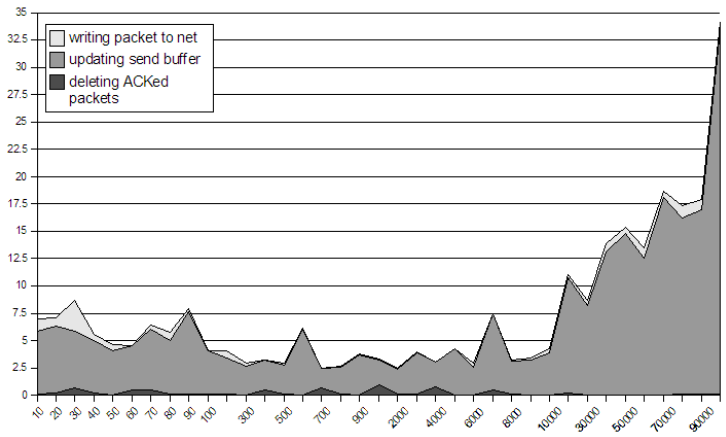

Figure 4.3: Processing Times, Sender

of packets take up. To that end, the code was instrumented with debug output at specific locations. Specifically, the following measurements were taken:

- Time to read a packet from the network or write it to the network (receiver and sender, respectively)
- Time to build or read the header, and update the packet with the header information
- Time to update the buffer and related data structures
- Time to persist the change
- Time to push packets up to the application, and remove them from the buffer (receiver only)
- Time to send the ACK (receiver only)

The measurement results are depicted in figures 4.3 and 4.4. What the figures show clearly is that the lion's share of the processing time is spent when the transaction is committed to disk. Note that each commit comprises two disk writes, one for the write-ahead log, and one when the object is persisted. Considering that even fast hard drive have random seek times of around 8 milliseconds, the minimal latency of 5 milliseconds is quite acceptable. (The disk writes do not occur at random places, hence the access times can be faster than aforementioned 8 milliseconds). Several measured points (building the header, updating the packet, updating the data structures) took such insignificant amounts of time that they could not be measured within a millisecond accuracy and are omitted in the figures.

Testing the RTO Algorithm The results of the previous two tests are encouraging, since they are a strong cue that the RTO algorithm, which produced accurate results in the simulation, also gives good results in practice. The next test was set up to confirm this belief. The same 3 scenarios as in 2.3.4 have been tested:

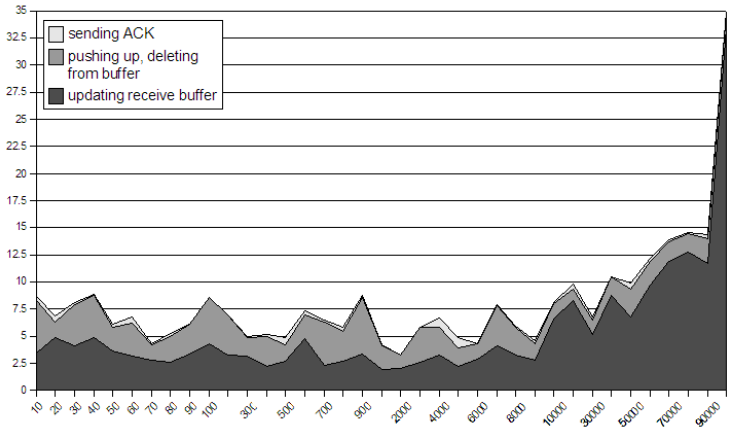

Figure 4.4: Processing Times, Receiver

- A constant minimal round-trip time of 2000 milliseconds. This test setup is designed to investigate how fast the RTO algorithm (or, more precisely, it's initial value) converges to a steady state.
- A steady change of the minimal round-trip time from 2000 milliseconds to 3000 milliseconds. The goal of this test is to find out how the algorithm reacts to slow changes of the RTT.
- An abrupt change of the minimal RTT from 2000 to 3000 milliseconds, to test the reaction to sudden changes in the network connection.

In all three test scenarios, packages with uniformly randomly selected sizes between 15 bytes and 15 kilobytes were sent. The specified delays have a standard deviation σ of 10%. [3]The link delay has been set to a very large value because the RTO algorithm demands a minimum value of 1 second, and with smaller delays would not result in an RTO above this lower bound for the tested packet sizes.

Because the processing delay of the ReTCP layer is negligible for the chosen packet sizes and link delays, the bandwidth was also limited to 15kB/sec. This introduces an additional delay between 0 and 1 seconds depending on the packet size and simulates the processing delay of a slow host. Without this bandwidth limitation, the packet size would have almost no influence on round-trip times, and the test would simply test the standard TCP RTO algorithm. Figures 4.5 trough 4.7 show the calculated RTO normalized to MSS-size (1460 bytes).

The figures are in very good accordance with what was to be expected from the simulation. The initial convergence to the constant RTT occurs after 10 to 15 received ACKs (figure 4.5). The ReTCP algorithm reacts somewhat more pronounced to the steady change of the RTT than TCP's RTO would; the graph shows a few noticeable spikes. This is to be expected, as the change of

[3]Instead of a normal distribution of the delays, NIST Net by default uses a "heavy tail" distribution which more closely models the delay patterns found in real networks. This default distribution was used, so the test somewhat differs from the simulation, but is executed in a more realistic environment.

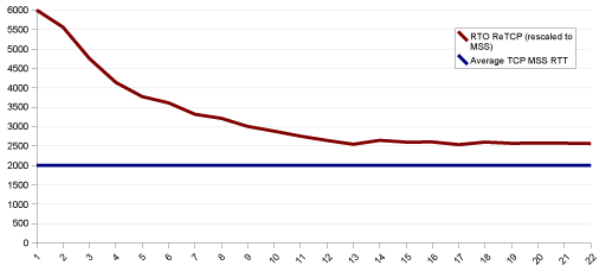

Figure 4.5: Convergence to constant RTT

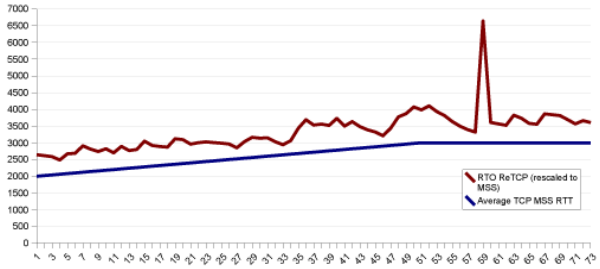

Figure 4.6: Slow RTT increase from 2 to 3 seconds

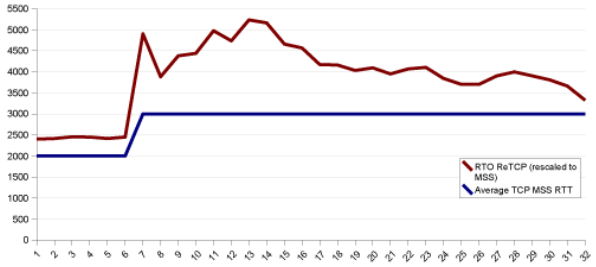

Figure 4.7: Abrupt RTT change from 2 to 3 seconds

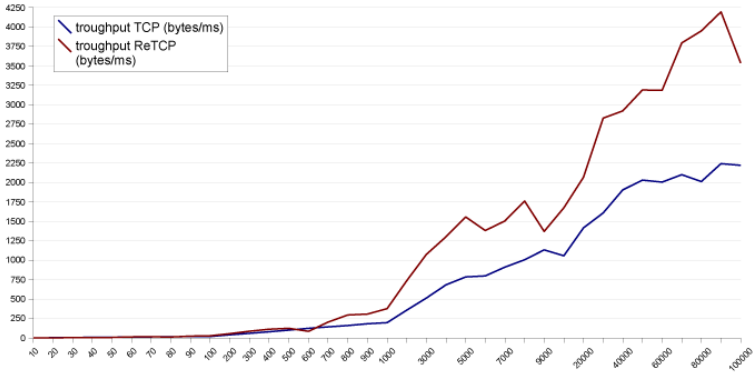

Figure 4.8: Throughput, Ethernet

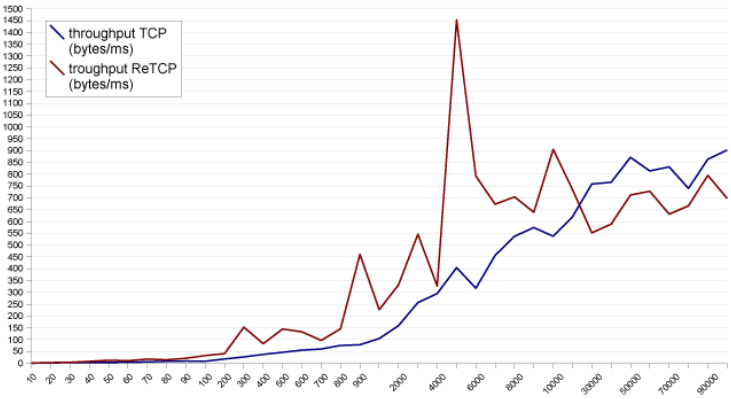

Figure 4.9: Throughput, Wifi-5

the RTT induces a change of the scaling variance, and the broad range of tested packet sizes amplifies this variance. The RTO quickly converges to a stable value again once the RTT no more changes. (figure 4.6). In the last figure (4.7), the exponential backoff of the RTO is clearly visible. Again, after around 20 received ACKs the RTO has converged to a steady state.

4.2.3 Throughput

Throughput was tested under under the 3 scenarios *Ethernet*, *Wifi-5*, and *Wifi-10* as defined above. The test is performed by sending packets of sizes ranging from 10 bytes to 100 kilobytes from a publisher to a broker, over a link with the emulated characteristics defined in table 4.1. For each packet size, 1500 packets of that size are sent. Debug code is added to the broker that keeps track of the amount of received packets, their size, and the elapsed time. Figures 4.8 to 4.10 show the measurement results. Please not that the x-axis is not scaled linearly.

The measurement results for the Ethernet case are pretty similar for ReTCP and QoS 2. This is not surprising: since the cost of persisting a message is relatively high, as it involves several disk writes, and the latency of the link is

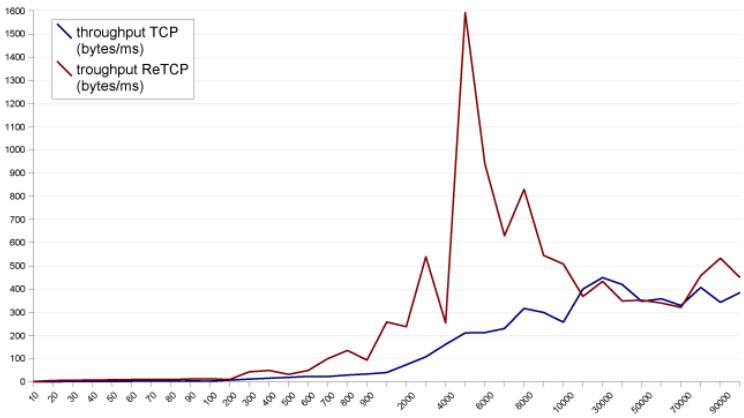

Figure 4.10: Throughput, Wifi-10

almost negligible, the time needed for disk writes prevails and the additional round-trip times the QoS 2 protocol needs, in which time no data packets can be sent, do not greatly affect the measurement result.

The situation changes when loss is present on the network. The publisher of a QoS 2 message has to wait 2 round-trip times (until the PUBCOMP is received) before it can send the next message. The higher the error probability of the channel is, the more likely it is that 1 of the 4 messages involved in a QoS 2 publication gets dropped on the TCP-level and will be resent by TCP when it's RTO expires. Together with the already larger initial link delay, this greatly increases the time during which the publisher is completely idle, waiting for the outstanding PUBACK. In the Wifi-5 scenario, this effect is already clearly obvious; in the Wifi-10 scenario, it is even more pronounced.

Clearly, sequentially completing the exchange of 4 messages until the next message can be sent is a big drawback of the QoS 2 protocol, especially for high link delays and high channel error probabilities. The available bandwidth is highly underutilized, leading to poor throughput. ReTCP, as expected, provides better usage of the available bandwidth. For large packet sizes, the advantage of ReTCP is less distinctive; here the time needed to transmit a packet over the lossy channel dominate the effects of using different protocols.

4.2.4 Message Delivery Latency

The next test measures the delivery latency of messages. The delivery latency is the time when the sending application hands a message to the network stack until it is available to the receiving application. Note that this is different from the round-trip time. The reason for measuring delivery latencies rather than RTTs is that for real-time applications, the delivery latency is the important metric.

Good performance in terms of low delivery latency is perhaps more important for most MQTT applications, as most MQTT data sources are unlikely to produce data amounts suited to fully utilize the available bandwidth. Low latencies, on the other hand, are desired in many real-life applications.

Again, the three test scenarios described before are used. To measure the

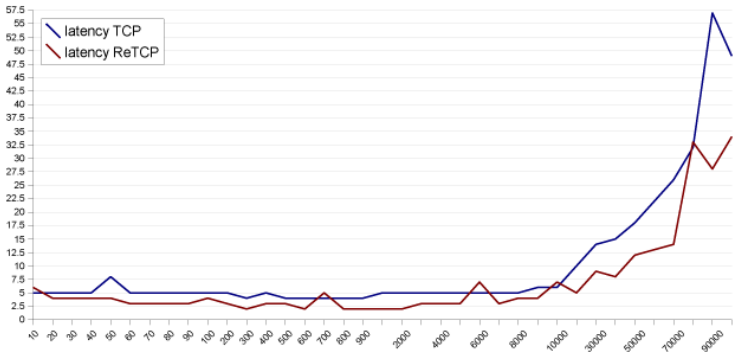

Figure 4.11: Delivery latency (ms), Ethernet

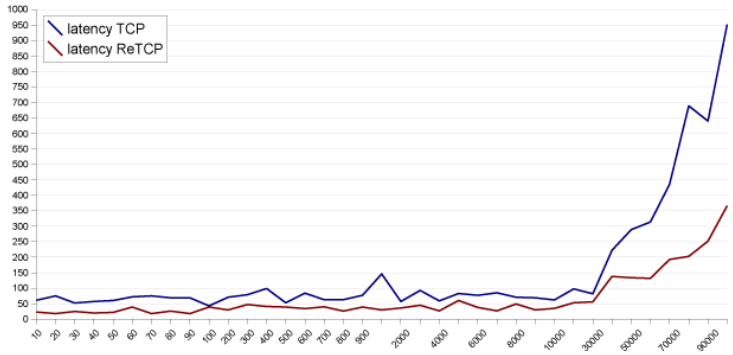

Figure 4.12: Delivery latency (ms), Wifi-5

time the message needs to get from the publisher application to the broker application, the publisher places a milliseconds-timestamp in the message. Since both broker and publisher run on the same machine, the publisher can immediately deduce the amount of passed time from the timestamp by comparing it with its own clock. Once again, for each message size the test is executed 1500 times, and the average delivery delay measured. Figures 4.11, 4.12, and 4.13 show the results.

The results are as expected: the ReTCP delivery latency is generally below the one of QoS 2. This is not surprising, as QoS 2 needs one RTT more until the message is delivered. In the Ethernet case, the difference between the latencies is fairly small. This is because the emulated Ethernet link has virtually no delay. For the two Wifi scenarios, the there is a noticeable difference between the latencies.

In all 3 cases, the test confirms the assumption that ReTCP performance in terms of delivery latency is superior to the one of the QoS 2 protocol.

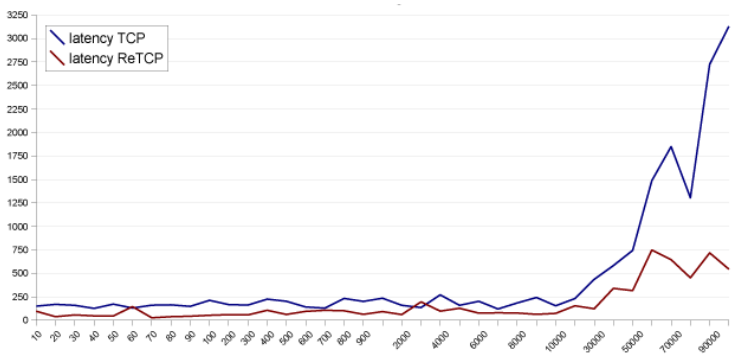

Figure 4.13: Delivery latency (ms), Wifi-10

Chapter 5

Conclusion and Future Work

5.0.5 Future Work

This section is a list of possible future work based on the results of this master's thesis.

- A generic ReTCP implementation.
  While the presented ReTCP protocol is useful for many different applications, an implementation is only provided in the context of the MQTT network stack. A general-purpose implementation providing an easy-to-use interface for applications wishing to communicate in a failure-resilient way is desirable. This implementation could for example be written as a C-based kernel module.

- UDP-based ReTCP protocol extension.
  ReTCP uses many of the same mechanism as TCP. Most notably, packets are identified by unique sequence numbers defining an ordering, and packets are resent when an acknowledgment is not received in a timely manner. To some degree (namely, whenever the underlying network and the applications do not fail), the work performed by ReTCP and TCP is redundant. Therefore, an extended ReTCP protocol combining the features of TCP and UDP could be designed. This protocol could use UDP as the underlying transport mechanism.
  Because ReTCP explicitly relies on TCP's flow control and congestion avoidance mechanisms, these mechanisms must be added to a UDP-based ReTCP protocol.

- End-to-end flow control for MQTT.
  Currently, when QoS 1 or QoS 2 messages cannot be delivered to a recipient because the recipient cannot receive them fast enough, the broker adds them to a queue in memory. This can lead to a situation where the broker needs more memory than is available. It cannot simply throw away messages in that case, because it must guarantee delivery for messages with a QoS level above 0. This situation is currently not addressed; a broker can be crashed by overwhelming it with messages for a slow receiver.
  When using ReTCP, a broker can write the messages to the ReTCP level, and they will be stored in the outgoing buffer. For a slow receiver, the

buffer will eventually become full, and further attempts to add packets to it will block.
Instead, a notification mechanism could be added allowing the ReTCP layer to inform the broker that the buffer is running full. Several ways how the broker then reacts to this situation are conceivable: it could disconnect receivers that are too slow, disconnect senders that are too fast, or even inform a fast sender that it should transmit new messages at a lower rate.

5.0.6 Conclusions and Retrospection

Designing and implementing a persistency layer for MQTT was a challenging, but also very rewarding and instructional task. The problems encountered when designing a network protocol are manifold, and not all of them are obvious. Perhaps the most intriguing problem that posed itself was the design of an RTO algorithm that computes accurate predictions of round-trip times for packets with sizes in the range of many orders of magnitude; a problem that was overlooked until a very late phase of the design process.

Some routes taken during the past 6 months also turned out to be dead ends. For example, the earliest implementation of ReTCP used a selective acknowledgment mechanism. The SACK algorithm is not only more complex to implement and computationally more expensive, it is also useless in most cases since TCP already provides in-order delivery. The only time when a selective acknowledgment is useful for ReTCP is when a ReTCP-session is reestablished; however, the fast retransmit algorithm solves this problem in a much simpler way.

The provided implementation fulfills the requirements defined during the design phase. Several experiments have been carried out, that showed that the implementation indeed behaves as expected and provides increased throughput as well as lower delivery latencies compared to the currently used protocol. Integration tests proved that ReTCP can deal with network failures as well as application failures, and no packets are lost once an application has handed them to the ReTCP stack module.

Choosing emulation over simulation for the performed experiments was a good choice, as it provides a higher level of confidence in the measured results. By using emulation, not only the protocol was tested, but also the actual implementation, including the effects of many concurrent disk-writes that a simulation can hardly model accurately.

Moreover, applying performance tests to the real code in fact yielded several performance improvements. Most notably, an early implementation of the `StackState` class simply contained 2 `ArrayLists` of tokens representing representing the send- and the receive-buffer, respectively. The entire object, and hence both buffers, were persisted when one of the buffers was updated. This increased the amount of data written to disk for each update, and a newer implementation that persists the buffers independently of each other shows better results in the performance tests.

The previous section lists some possible extensions and improvements to the presented ReTCP implementation. I hope that the ideas and results provided by this thesis inspire further development of this interesting project.

Appendix A

RTO Calculation

A.1 TCP RTO Calculation

The RTO (retransmission timeout) algorithm describes how a peer calculates its retransmission timeout. The choice of particular parameters is beyond the scope of this document; they are suggested by the research of Jacobson et al. ([JK88]).

2 state variables, $SRTT$ (smoothed round-trip time) and $RTTVAR$ (round-trip time variance) are maintained. G refers to the clock granularity of the implementation, which is 10 milliseconds in the implemented prototype.

Algorithm 1 shows the RTO calculation, and algorithm 2 shows how the retransmission timer is managed using the calculated RTO.

A.2 ReTCP RTO Calculation

Algorithms 4 and 3 define how the RTO is for different packet sizes is calculated and updated in ReTCP.

Algorithm 1 RTO Calculation, as per RFC 2988.

```
1. RTO ← 3 seconds
2. when the first RTT measurement R is made:
        SRTT ← R
        RTTVAR ← R/2
        RTO ← SRTT + max (G, K*RTTVAR)
   where K=4
3. for each subsequent measurement R':
        RTTVAR ← (1 - beta) * RTTVAR + beta * |SRTT - R'|
        SRTT ← (1 - alpha) * SRTT + alpha * R'
        RTO ← max (1 second, SRTT + max (G, K*RTTVAR))
   where alpha=1/8, beta=1/4
```

Algorithm 2 Retransmission timer management, as per RFC 2988

```
1. Every time a packet containing data is sent (including a
   retransmission), if the timer is not running, start it
   running so that it will expire after RTO seconds (for the
   current value of RTO).
2. When all outstanding data has been acknowledged, turn off
   the retransmission timer.
3. When an ACK is received that acknowledges new data,
   restart the retransmission timer so that it will expire
   after RTO seconds (for the current value of RTO).

When the retransmission timer expires, do the following:
4. Retransmit the earliest segment that has not been
   acknowledged by the TCP receiver.
5. The host MUST set RTO ← RTO * 2 ("back off the timer").
   A maximum value ay be used to provide an upper bound to
   this doubling operation.
6. Start the retransmission timer, such that it expires after
   RTO seconds (for the value of RTO after the doubling
   operation outlined in 5.).
```

Algorithm 3 ReTCP RTO Scaling

```
function scaleRTO(packetSize) {
    # estimates the RTO for a packet with the given size,
    # based on previous RTT measurements for various packet
    # sizes
    dist ← packetSize - SSIZE
    r ← RTO + dist * SSCALE + |dist| * SCALEVAR
    if (dist < 0); then
        return max( 1000, min(r, RTO, RTO_MAX) )
    else
        return max( 1000, min(r, RTO_MAX) )
    fi
}
```

Algorithm 4 ReTCP RTO Calculation

```
#variables:
# RTO: retransmission time-out
# SRTT: smoothed round-trip time
# RTTVAR: round-trip time variance
# SSIZE: smoothed packet-size
# SSCALE: smoothed scaling factor
# SCALEVAR: scaling factor variance
#constants:
# K: weighting of variance
# ALPHA, BETA: smoothing factors for RTO, variance
# RTO_MAX: upper bound for RTO

1. RTO ← 6000
   SSIZE ← 1
   SSCALE ← 3000
   SCALEVAR ← 3000

2. when the first RTT measurement R is made, for
   packet with size p:
       SSCALE ← R/p
       SCALEVAR ← SSCALE/2
       SSIZE ← p
       SRTT ← R
       RTTVAR ← R/2
       RTO ← min( RTO_MAX, SRTT+max(G, K*RTTVAR) )
   where K=4

3. subsequent RTT measurements R', for packet
   with size p:
       if (p ≠ SSIZE); then
           # update the scaling factor
           s ← (R' - SRTT) / (p - SSIZE)
           s ← max( 0, min(SRTT/SSIZE+SCALEVAR, s) )
           SCALEVAR ← (1 - BETA) * SCALEVAR + BETA * |SSCALE - s|
           SSCALE ← (1 - ALPHA) * SSCALE + ALPHA * s
       fi
       # scale the measurement R' according to scaling factor,
       # and use scaled value to update RTO
       R̂ ← R' - (p - SSIZE) * SSCALE
       RTTVAR ← (1 - BETA) * RTTVAR + BETA * |SRTT - R̂|
       SRTT ← (1 - ALPHA) * SRTT + ALPHA * R̂
       r ← SRTT + max(G, K*RTTVAR)
       RTO ← max( 1000, min(r, RTO_MAX) )
   where ALPHA=1/8, BETA=1/4
```

Bibliography

[MQTT] *MQ Telemetry Transport.* http://www.mqtt.org/

[MQ] *WebSphere MQ.* http://www-306.ibm.com/software/integration/wmq/

[norw] *Pay-as-you-drive car cover tested.* http://news.bbc.co.uk/2/hi/business/3573912.stm. BBC News, August 2004.

[RFC793] *Transmission Control Protocol.* RFC 793, September 1981.

[MQTT-spec] *MQTT Protocol Specification.* http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp?topic=/com.ibm.etools.mft.doc/ac10840_.htm

[RAI99] Rajiv Chakravorty, Andrew Clark, Ian Pratt: *GPRSWeb: Optimizing the Web for GPRS Links.* University of Cambridge Computer Laboratory, 1999.

[SHB+04] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, Mat Welsh: *Simulating the Power Consumption of Large-Scale Sensor Network Applications.* ACM 1-58113-879-2/04/0011.

[Per05] Julio Perez: *MQTT Performance Analysis with OMNeT++.* Master's Thesis, September 2005.

[SS83] Dale Skeen, Michael Stonebraker: *A Formal Model of Crash Recovery in a Distributed System.* IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, may 1983.

[LSP82] Leslie Lamport, Robert Shostak, Marshall Peas: *The Byzantine Generals Problem.* ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, pp 382-401, July 1982.

[HOI+05] Osamu Honda, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka, Junichi Murayama: *Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency.* Proceedings of the SPIE, Volume 6011, pp. 138-146, 2005.

[Titz01] Olaf Titz: *Why tcp over tcp is not a good idea.* http://sites.inka.de/~bigred/devel/tcp-tcp.html

[RFC1122] Robert Braden: *Requirements for Internet Hosts - Communication Layers.* RFC 1122, October 1989.

[RFC896] John Nagle: *Congestion Control in IP/TCP.* RFC 896, January 1984.

[RFC2988] V. Paxson, M. Allman: Dale Skeen, Michael Stonebraker: *Computing TCP's Retransmission Timer.* RFC 2988, November 2000.

[JK88] V. Jacobson, M. Karels: *Congestion Avoidance and Control.* ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[KP87] P. Karn, C. Partridge: *Improving Round-Trip Time Estimates in Reliable Transport Protocols.* ACM Transactions on Computer Systems, Vol. 9, No. 4, pp. 364-373, 1991.

[RFC879] J. Postel: *TCP maximum segment size and related topics.* RFC 879.

[xkernel] *The x-kernel Protocol Framework.* http://www.cs.arizona.edu/projects/xkernel/

[omnet] *OMNeT++ Discrete Event Simulation System.* http://www.omnetpp.org/

[NISTNet] National Institute of Standards and Technology: *NIST Net Home Page.* http://www-x.antd.nist.gov/nistnet/

[CD03] Mark Carson, Darrin Santay: *NIST Net – A Linux-based Network Emulation Tool.* Computer Communication Review, June 2003.