Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Concept and Implementation of TCP for the RIOT Operating System and Evaluation of Common TCP-Extensions for the Internet of Things

Simon Brummer

Bachelorthesis

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Simon Brummer

Concept and Implementation of TCP for the RIOT Operating System and Evaluation of Common TCP-Extensions for the Internet of Things

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Schmidt
Zweitgutachter: Prof. Dr. Hans Heinrich Heitmann

Eingereicht am: 1. April 2016

Simon Brummer

Thema der Arbeit

Concept and Implementation of TCP for the RIOT Operating System and Evaluation of Common TCP-Extensions for the Internet of Things

Stichworte

TCP, RIOT, Internet der Dinge, Netzwerke

Kurzzusammenfassung

Das Transmission Control Protocol (TCP) bildet die Basis der meisten Anwendungen im Internet. Viele Protokolle benötigen den von TCP bereitgestellten, zuverlässigen Datentransport. Im Internet der Dinge dominiert das Transportprotokoll UDP, da den eingesetzen Computern meist nur geringe Hardwareressourcen zur Verfügung stehen. Das Betriebssystem RIOT ist für den Einsatz im Internet der Dinge konzipiert. Es entsteht derzeit ein neuer, modularer Netzwerkstack für RIOT, der alle gänigen Netwerkprotokolle unterstützen soll, somit auch TCP. Im Rahmen dieser Bachelorarbeit wird die Eignung von TCP für das Internet der Dinge untersucht werden. Desweiteren wurde TCP für den neuen Netzwerkstacks von RIOT implementiert sowie verbreitete TCP Erweiterungen auf ihre Eignung im Internet der Dinge diskutiert.

Simon Brummer

Title of the paper

Concept and Implementation of TCP for the RIOT Operating System and Evaluation of Common TCP-Extensions for the Internet of Things

Keywords

TCP, RIOT, Internet of Things, Networks

Abstract

The Transmission Control Protocol (TCP) is the basis for most applications on the Internet. Many protocols depend on the reliable data transport offered by TCP. The Internet of Things is dominated by UDP as transport protocol due to the lack of resources of the computers in use. The RIOT operating system is designed for the Internet of Things. Currently there is a new modular network stack developed for RIOT. It should support most common network protocols including TCP. In this bachelor thesis, TCP is evaluated for its suitability for the Internet of Things. Additionally TCP is implemented for the new RIOT network stack and common TCP Extensions are reviewed for their applicability in the Internet of Things.

Table of Contents

List of Tables

List of Figures

List of Code listings

1 Introduction

The Internet of Things arrives more and more in our daily lives, leading to a significant paradigmatic change in network technology. We see a trend to connect more and more embedded devices to the Internet, sharing information among people, as well as other machines. This enables developers from all over the world to create new applications and optimize existing technologies. Embedded systems range from mobile phones, health monitor devices, home automation and industrial automation, smart metering to environmental monitoring systems.

The Internet core protocols IP, UDP and TCP are the building blocks for interconnecting the classical infrastructure and new mobile embedded devices. Existing networks were characterized by nodes with high processing power, stationary power supply, fast, reliable, often wired network connections. Most of the core networking protocols were designed with the classical infrastructure in mind, but with the rise of low-power embedded devices, network protocols need to operate in a different environment. Unreliable, battery-powered, non-stationary nodes with low-power wireless communication and low processing power are common in IoT scenarios.

The RIOT operating system is aimed for class 1 constrained devices [BEK14, p.8]. It provides extensive network capabilities to connect embedded devices on the Internet. Currently the RIOT network stack is reimplemented to be more generic and modular compared to the previous implementation.

In this thesis, the transport protocol TCP is analyzed and implemented for the new RIOT network stack with respect to the demands of current IoT technology. Furthermore, common TCP extensions will be reviewed for suitability in IoT-Scenarios.

1.1 Motivation

The new RIOT network stack was released without TCP support, limiting the applicability of RIOT. However, TCP is the most common transport protocol on the Internet today. By supplying a basic TCP implementation, a large numbers of protocols which rely on TCP can be ported and used in IoT scenarios. A TCP implementation allows reliable communication between

IoT nodes and general purpose computers on an application level, facilitating interconnection between applications in general.

1.2 Organization

Background information on common IoT technologies and RIOT-OS is the topic of Chapter 2. This Chapter covers protocols typical for IoT scenarios, like 6LoWPAN, and common network architectures for embedded devices. Additionally, this chapter provides a short overview of the RIOT operating system, especially the architecture and technical details of the new generic network stack (gnrc).

Chapter 3 explains the basic mechanisms of the transmission control protocol. This includes the TCP core concepts and applications, header content, connection establishment and termination, data transfer and receive window management. This chapter conveys a solid understanding of the TCP core concepts and how they interact with each other.

Over the years, the basic TCP standard has been extended and improved to adapt developments in computer networks. In Chapter 4, selective acknowledgment options and congestion control mechanisms will be explained and analyzed for their usability in IoT scenarios. These extensions are a common part of every mature TCP implementation.

In chapter 5, µIP and lwIP, two TCP/IP stacks designed and deployed on embedded devices are analyzed. Their basic characteristics are compared and displayed.

Chapter 6 focuses on the concepts of the new TCP implementation for RIOT. In this chapter, design goals for the TCP implementation are defined, as well as means to achieve them. The design goals reach from the avoidance of dynamic memory allocation to seamless integration into the gnrc network stack.

The topic of Chapter 7 is the TCP implementation for RIOT. This chapter covers important aspects of the TCP implementation in relation to the design goals defined in of Chapter 6.

Testing and verification is an important topic in every protocol implementation. Chapter 8 covers testing methodology and devised test scenarios for the TCP implementation. The measured results are explained in detail.

The final chapter summarizes the thesis. Conclusions are drawn about TCP in the context of the Internet of Things and future goals for improvement of the TCP implementation are formulated.

2 Background information on the IoT and RIOT-OS

The first section in this chapter gives a definition and an overview of the wireless embedded Internet. It includes specialized architectures designed for IoT usage, differences between the traditional and embedded network stacks and an overview of 802.15.4 and 6LoWPAN as common protocols for embedded scenarios. The second section introduces the RIOT operating system in general, and the core principles behind the new network stack named "gnrc".

2.1 Wireless embedded Internet

The Internet of Things is a widely used term to encompass all embedded devices that are Internet-connected and IP-enabled. Often, those devices are controlled and monitored by external services. The wireless embedded Internet is a subset of the IoT. It consists mostly of constrained, often battery powered devices connected by low-power, low-bandwidth wireless networks to the Internet. The core technology behind network interconnection is IPv6 [DH98]. IPv6 features extended addressing capabilities, simplified IP headers and improved support for extensions and options, as well as protocols for neighbor discovery [NNSS07] and address auto-configuration [TNJ07]. Optional IPv6 features like IPSec to support authentication and encryption on a network layer or a MTU of 1280 bytes are demanding for embedded devices. This led to the idea of adaptation protocols like 6LoWPAN [MKHC07], to adapt IPv6 and linked protocols (e.g. neighbor discovery) for usage in constrained devices. These adaptation protocols are usually part of embedded IP stacks and require specialized network architectures for wireless nodes.

2.1.1 The 6LoWPAN architecture

According to Zack Shelby and Carsten Borman, "the Wireless Embedded Internet is created by connecting islands of wireless embedded devices, each island being a stub network on the Internet. A stub network is a network which IP packets are sent from or destined to, but which

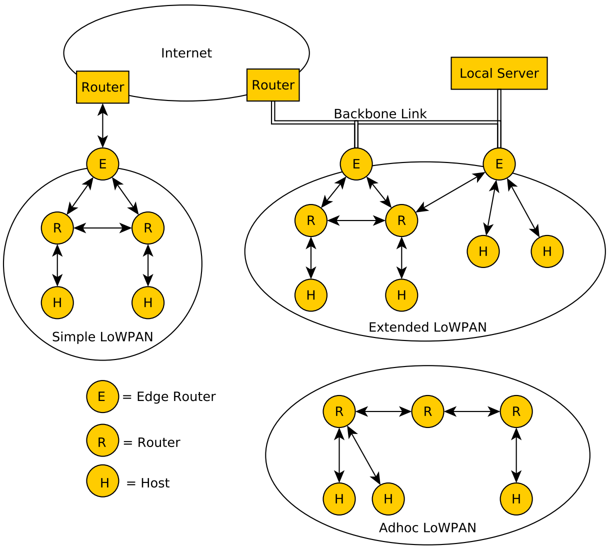

Figure 1: 6LoWPAN architectures, see [SB09, p.14]

does not act as a transmitter to other networks“ [SB09, p.13]. The 6LoWPAN architecture consists of LoWPANs that share a common IPv6-prefix. Interconnection between a 6LoWPAN and other networks is achieved via edge routers. Figure 1 shows three common network topologies for 6LoWPAN.

Three different types of LoWPAN networks are defined:

- An ad hoc LoWPAN, that is not connected to other networks and operates autonomously.
- A simple LoWPAN, connected to other IP-Networks via one edge router.
- An extended LoWPAN, connected via multiple edge routers to a common backbone link.

The role of edge routers is important in those topologies. They are the gateways between a LoWPAN and other networks. Edge routers handle routing, adaptation between IPv6 and 6LoWPAN on incoming and outgoing traffic, neighbor discovery for the LoWPAN and other network management features. The nodes in a LoWPAN are hosts, edge routers or nodes routing between other nodes. The nodal network interfaces in a 6LoWPAN share a common IPv6-prefix, which is advertised by edge routers and routers through the LoWPAN or is configured in advance on each node. An edge router maintains a list of registered nodes, reachable via its own network interface within the LoWPAN.

A node joins a 6LoWPAN by sending a Router Solicitation message to receive the IPv6-prefix of this LoWPAN, if not statically configured. After receiving the prefix, a unique global IPv6-address is built. The node registers this address at the edge router of this LoWPAN. The edge router now has the information needed for routing decisions in and out of the LoWPAN and information needed for the 6LoWPAN neighbor discovery. Additionally, edge routers handle header compression and decompression on network traversal. The list of nodes must be refreshed on a regular basis because registered addresses expire after a configurable amount of time. A rather long expiration time reduces power consumption of a node, a short expiration time allows fast changing network structures. These operations are part of the specialized neighbour discovery mechanism for 6LoWPAN [SCNB12]. LoWPAN nodes are free to move inside and between multiple LoWPAN networks and they can be part of multiple LoWPAN at the same time. Communication between a LoWPAN node and an external IP node happens in an end-to-end manner just as between normal IP nodes.

For example, a node moving between two simple LoWPANs tries to refresh the address entry at its edge router, where the node registered itself previously. By leaving the radio range of the LoWPAN where the node was registered, the old edge router is unreachable. The node reacts by sending a new Router Solicitation message to attain a new IPv6-prefix. On reception of the IPv6-prefix, the node builds a new address and registers at the corresponding edge router of the reachable LoWPAN.

The previously registered edge router entry expires after some time, removing old routing and neighbor discovery information from the LoWPAN. The node moved successfully between the two LoWPANs.

In an extended LoWPAN, multiple edge routers are part of the same LoWPAN, propagating the same IPv6-prefix. The edge routers are connected via a shared backbone link. A node moving between edge routers still needs to register at the edge router it can reach but the node can keep its IPv6-address. The messaging between edge routers related to neighbor discovery is offloaded onto the backbone link, reducing the messaging overhead. The extended LoWPAN architecture enables a single LoWPAN to span over large areas.

A LoWPAN can operate without a connection to other external networks as well. This is called ad hoc LoWPAN. Only one node needs to act as a simplified edge router. This node must generate a unique local unicast address and it needs to supply neighbor discovery registration functionality. It works like a simple LoWPAN without the link to other external IP-based networks.

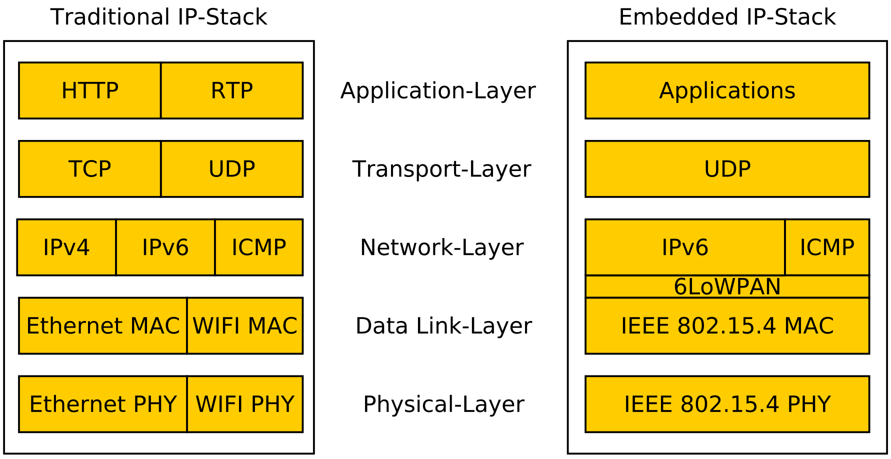

Figure 2: Differences between traditional and embedded IP-Stack, see [SB09, p. 16]

2.1.2 Embedded IP stack

The traditional inter-network stack has grown over the last decades. It supports a large amount of existing technologies and resource demanding network standards. These stacks are designed for general purpose operating systems, offering more features than needed in an Internet of Things context. Therefore, the goal behind the development of embedded IP-stacks is the focus on the essential protocols necessary for network operation.

Figure 2 shows common differences between traditional and embedded IP-stacks in a simplified version of the OSI model. The embedded IP-stack is a reduced version of the traditional stack. Most embedded nodes only have one network interface except edge routers. Therefore only one technology below the link-layer needs to be supported. The 6LoWPAN adaptation layer is situated between link- and network layer. 6LoWPAN is necessary to enable IPv6 operation over lossy low-power link-layer technologies like IEEE 802.15.4. On the network layer IPv6 and ICMPv6 are usually supplied, superseding IPv4 usage in general. On the transport layer, UDP is favored over TCP due to the inherent complexity of TCP. It is common for applications to use binary protocols instead of human readable standards to minimize the amount of data that needs to be transmitted.

Edge routers play an important role in enabling connectivity between traditional and embedded network stacks. They ensure traversal between different link-layer technologies and routing, therefore they need to handle multiple network interfaces and technologies present in

Frequency range (MHz)	Region	Channel numbers	Bit rate (kbits/s)
868	Europe	0	20
902-928	US	1-10	40
2400-2483.5	Worldwide	11-24	250

Table 1: Frequency ranges and channels for IEEE 802.15.4, see [SB09, Appendix B.1].

traditional and embedded stacks up to the network layer. Edge routers handle the conversion between full IPv6 and the 6LoWPAN format for in- and outgoing traffic. 6LoWPAN features stateless and stateful header compression algorithms. IPv6- and UDP headers, for example can be compressed by 6LoWPAN by omitting information known to every node in the 6LoWPAN. Nodes inside the LoWPAN are able to restore these compressed headers. For packets with destinations outside the LoWPAN, an edge router must translate compressed IPv6 headers to normal IPv6 headers before routing them to another network.

2.1.3 IEEE 802.15.4

The IEEE 802.15.4 standards are specified by the IEEE for low-power wireless radio techniques. IEEE 802.15.4 specifies the physical and media access control-layers and is the foundation 6LoWPAN is build upon.

The latest version of the standard IEEE 802.15.4-2011 [Soc11] features access control via CSMA/CA, optional acknowledgments for retransmission of distorted data, as well as 128-bit AES encryption on the link-layer. "Addressing modes utilizing 64-bit and 16-bit addresses are provided with uni and broadcast capabilities. The payload of a physical frame can be up to 127 bytes in size, with 72-116 bytes of payload after link-layer framing, depending on a number of addressing and security options"[SB09, Appendix B.1].

IEEE 802.15.4 supports star and point-to-point network topologies. The MAC-layer operates either in a beacon-less or a beacon-enabled mode. The beacon-less mode uses CSMA/CA, the beacon-enabled mode uses TDMA/TISCH for media access.

Table 1 shows that IEEE 802.15.4 radios are divided into two regional and a worldwide frequency range, they differ in available channels and bit rate.

According to [SSZV07], the average packet loss of IEEE 802.15.4 depends heavily on the number of nodes, the transmitted message length and current radio interference in the used ISM band. In terms of reliability, the usage of acknowledgments on the link-layer improves reliability but makes packet round trip times hard to estimate.

2.1.4 6LoWPAN

The purpose of the IP protocol is to interconnect networks, independent of the underlying network technologies. The link-layer technologies often differ between traversed networks, therefore each type of network needs an "IP-over-X" specification to define how IP is converted onto the underlying link-layer. Those specifications differ in complexity, e.g. IPv6-over-Ethernet [Cra98] is rather simple, because IPv6 is closely aligned with Ethernet. Other standards, like PPP [Sim94] require more work to map services that are needed for IPv6 operation onto a lower layer. This complexity can amount to an independent adaptation layer, like 6LoWPAN adapting IPv6 to IEEE 802.15.4.

6LoWPAN handles IPv6-packet fragmentation, header compression, multicasting and routing in mesh-networks. Fragmentation and reassembly is necessary because IPv6 demands a maximum transmission unit (MTU) of 1280 bytes. This means the layer below IPv6 must be able to transmit a 1280 bytes payload within one packet. On the other hand, IEEE 802.15.4 transmits only 127 bytes frames on a physical layer. To solve this contradiction, 6LoWPAN handles fragmentation and reassembly to map IPv6-datagrams onto IEEE 802.15.4 frames.

Another issue, related to the small frame size of IEEE 802.15.4 is header compression. Every uncompressed frame contains an IEEE 802.15.4 header, a 6LoWPAN dispatch byte, uncompressed IPv6 header fields and a transport protocol header, depending on the protocol in use. The header sizes often vary depending on used options and consume a considerable amount of the available frame size per packet. 6LoWPAN header compression decreases the amount of transmitted data by compressing the IPv6 header and the transport protocol header if possible. Header compression is currently standardized for IPv6 and UDP [HT11]. An IP header can be compressed by omitting information already known by every node of a 6LoWPAN, the "Version" field in IPv6 header for example, can be omitted. The version field value will always be 6 because only IPv6 is supported by 6LoWPAN. Transport protocols like UDP can be compressed by reducing the available space for port numbering. UDP header compression assumes the source and destination port numbers between 61616 (0xF0B0) and 61631 (0xF0BF). If a port in this range is used for each port number, only the four least significant bits need to be transmitted. The uncompressed port number can be restored by adding 616161.

IPv6 demands a multicast mechanism [Dee89] for the neighbor discovery mechanics [NNSS07]. Multicast is assumed to be provided by the link layer. IEEE 802.15.4 itself does not define multicast capabilities like e.g. Ethernet does. The inherent problems associated with multicast in mobile devices are explained in detail in [SWF10]. To cope with the lack of multicast on IEEE 802.15.4, neighbor discovery has been optimized for 6LoWPAN usage [SCNB12]. The edge routers node registration table is used for neighbor discovery instead of multicast, avoi-

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	<2kB	<30 kB	Partial	No	Partial	Yes	Partial	Partial
Tiny OS	<1kB	<4kB	No	No	Partial	Yes	No	No
Linux	~1MB	~1MB	Yes	Yes	Yes	No	Partial	Partial
RIOT	~1,5kB	~5kB	Yes	Yes	Yes	Yes	Yes	Yes

Table 2: Characteristics comparison between Contiki, Tiny OS, Linux and RIOT, see [BHG+13]

ding the need for multicast usage during neighbor discovery inside a 6LoWPAN entirely. In 6LoWPAN networks without optimized neighbor discovery, the mesh-under routing protocols of 6LoWPAN can be used to mimic a multicast mechanism on the link-layer.

“6LoWPAN supports Mesh-Under routing protocols that provide multicasting capabilities. One simple, but rather inefficient way to provide multicasting is flooding: a node that wants to emit a multicast just sends it using the radio broadcast provided by IEEE 802.15.4; nodes that receive such a broadcast simply echo the multicast unless they have seen (and echoed) it before“ [SB09, p. 60].

Multicasting and broadcasting in wireless networks is energy intensive, therefore multicasting should be avoided inside the LoWPAN entirely.

2.2 RIOT operating system

“The friendly Operating System for the Internet of Things“ (RIOT) is specialized for usage in IoT scenarios. It is distributed under the LGPLv2.1 License. By using this License, RIOT is free and open software, usable by and distributable to everyone. The license allows RIOT to be linked together with proprietary software and enforces that RIOT is modifiable by the end users[1].

The design objectives of RIOT-OS include real-time and multithreading capabilities, energy-efficiency and a small memory footprint, as well as a uniform API independent from the underlying hardware. RIOT features a scalable modular micro kernel architecture to minimize the dependencies between the operating systems core and other system components. Programs for RIOT are either written in C or in C++, enabling the usage of existing libraries e.g. microCoAP[2] implementing CoAP[SHB14] or the C++ Actor Framework (CAF)[3][CHS14]. Configuration and usage of specific modules is achieved at compile time.

[1] https://github.com/RIOT-OS/RIOT/wiki/LGPL-compliancy-guide

[2] https://github.com/1248/microcoap

[3] https://github.com/actor-framework

Table 2 shows a feature comparison between Conkiti[4], Tiny OS[5], Linux[6] and RIOT[7]. These operating systems are common in the Internet of Things. The most notable features of RIOT are the real-time capabilities and multithreading support.

The multithreading capabilities enable RIOT developers to write modular, event-driven software. Synchronization between threads is usually achieved by the kernels message passing API although more traditional synchronization methods like semaphores or mutexes exist. The amount of threads is limited by the amount of available memory and by the chosen stack-size for each thread.

As a real-time operating system, "RIOT enforces constant periods for kernel tasks (e.g. scheduler run, inter-process communication, timer operations)." [BHG$^+$13]. Real-time systems are defined as systems that guarantee a response within specified time constrains. A prerequisite for real-time capabilities is a constant runtime for kernel specific tasks. This restricts the kernel facilities to the exclusive use of static memory allocation. Applications and external libraries may allocate memory dynamically, although this is discouraged. To ensure foreseeable runtime behavior, an applications memory footprint must be known at compilation time.

Another special feature of RIOT is its tick-less, priority based scheduler. Unlike most operating systems, RIOT does not simulate concurrent execution by switching threads based on a timer that expires periodically. Instead, context switches occur on an interrupt, a voluntary context switch (e.g. calling the sleep-function or waiting for the reception of a message) or as implicit context switch (functions that unblock a higher prioritized thread, e.g. the expiration of a timer).

RIOT supports various CPU architectures and prototyping boards. To get an impression on hardware used in the IoT, Atmel's SAM R21 Xplained Pro Evaluation Kit[8] for example, is build around a Cortex-M0 32-bit microcontroller with 256 kB ROM and 32 kB RAM. RIOT supports the SAM R21 and other popular platforms like Arduino and the STM discovery boards.

The network stack "gnrc" is an essential part in the current major release RIOT-2015.09[9]. The previous network stack was considered too monolithic and too hard to maintain by the RIOT community. The old stack was developed concurrently by multiple people without a uniform concept. The lack of unified interfaces between the layers made modularity, testability and extensibility hard to achieve. Additionally, most layers supplied their own buffers. This

[4] http://contiki-os.org
[5] http://tinyos.net
[6] http://www.linux.com
[7] http://riot-os.org
[8] http://www.atmel.com/tools/ATSAMR21-XPRO.aspx
[9] https://github.com/RIOT-OS/RIOT/releases/tag/2015.09

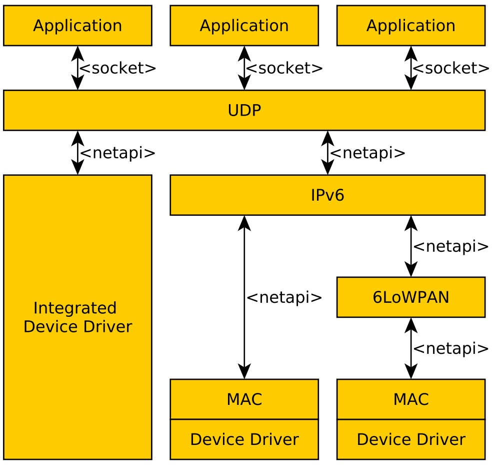

Figure 3: Typical network stack configuration, see [PLW+15]

increased the memory footprint and created the need for extensive copying between network layers, thus reducing overall performance. These drawbacks motivated the development of the "gnrc" network stack to supersede the existing one.

The design goals of the generic network stack include a low memory footprint, full-featured protocols, modular architecture, support for multiple network interfaces, parallel data handling and customization during compilation time.

By relying on the RIOT multithreading support, the communication between layers is achieved by IPC-mechanisms instead of fixed, interwoven function-calls. Every functional unit like UDP runs as a thread on RIOT, that communicates via message passing with the other modules. This approach leads to a less rigid network stack, allowing new modules to be integrated at every level of the network stack. The modular approach simplifies testing as well. Stub-layers can be written easily to verify inputs and outputs of a desired module, leading to more stable software in the long run. On the other hand, by relying on IPC accompanied by context-switches, function-calls and context-restorations, the modular approach suffers a performance penalty compared to simple function-calls. However, performance is not the dominating issue because network nodes in an IoT tend to communicate as little as possible, to save energy.

Figure 3 shows an example of a possible network stack configuration, each box is a module running in its own thread. Communication between modules is achieved by the netapi[10], an

[10] http://riot-os.org/api/netapi_8h.html

IPC-based API for communication between network modules. This example includes three network interfaces. The network stacks architecture separates device drivers and the layers of the network stack making modules reusable. The interface with an integrated device driver does not use the RIOT IPv6 implementation and communicates with the UDP layer directly. The second interface uses native IPv6 on the network interface, common in edge routers. The third interface communicates via IPv6 and the 6LoWPAN module which is typical for all 6LoWPAN nodes.

The generic network stack avoids repeated copying between layers by utilizing of a central packet buffer. "Outgoing data is copied from the user application (socket) into a central buffer and once into a networks interface's device buffer by the device driver. The same is true for received data, which is copied on arrival from a network interface into the central buffer and once more when handed over to an application." [PLW+15]. The central buffer is accessible from all network modules via an API called pktbuf[11].

The buffer provides memory for user data and header information. Packets inside the buffer are stored in a deduplication scheme, eliminating duplicate copies of whole packets or packet parts. To send a packet to another network module, only a pointer to the packet must be sent to another layer, instead of copying the whole packet between multiple buffers of different layers. By provision of the pktbuf API, the actual buffer implementation as well as allocated buffer sizes can be exchanged easily, e.g. the user could decide at compile time, to use a statically allocated buffer or a dynamically allocated buffer.

Additionally, the generic network stack features an API called netreg[12]. It serves as a central directory. During initialization, modules register in netreg with their thread ID and the kind of information they are interested in called "NETTYPE". A module responsible for IPv6 registers with its thread ID and type "NETTYPE_IPV6". An UDP module, for example passing a packet down the network stack, would use netreg to lookup the threads interested in type "NETTYPE_IPV6". The UDP module uses netapi, to send a pointer to the packet allocated in pktbuf to every thread registered in netreg on type "NETTYPE_IPV6".

These three new APIs are the building blocks for every layer of the new RIOT network stack, TCP is no exception. The next chapter covers the concepts behind TCP.

[11] http://riot-os.org/api/pktbuf_8h.html
[12] http://riot-os.org/api/netreg_8h.html

3 TCP overview

The transmission control protocol is the most widespread transport protocol on the Internet. Most standards requiring reliable data transport build on top of TCP. Widespread protocols like HTTP, FTP or SSH rely on TCP just to name a few. Basic TCP is specified in RFC 793 [Pos81] and has been extended numerous times. It provides a reliable, full-duplex, connection-oriented, ordered, error-corrected delivery of byte streams between two applications. Any byte stream can be transmitted via TCP without any restrictions. Interpretation of transmitted data is the task of an application and beyond the scope of TCP.

The basic TCP standard is rather complex. Its specification covers 85 pages without its various extensions. Full-featured operating systems supply TCP implementations. It is an indispensable protocol in today's computer networks.

3.1 TCP core concepts

TCP operations rely on a few basic concepts, namely basic data transfer, reliability, flow control, multiplexing and connection handling. Each concept covers an important aspect of TCP and is explained in this section.

Basic data transfer in TCP is specified as a full-duplex connection that is able to "transfer a continuous stream of octets in each direction between its users by packaging some number of octets into segments for transmission through the internet system" [Pos81, p.4]. The user hands over data to transmit, TCP decides on its own when to block and forward data, independent of the user. TCP provides a push function to cause TCP to deliver data as fast as possible. By pushing data from the sender to the receiver, TCP tries to forward and deliver data as fast as possible, up to the sequence number the push occurred. The receiver might not be aware of the exact push point.

Reliability is achieved by using sequence numbers and acknowledgments. In TCP each byte of data is assigned a sequence number. The sequence number of the first byte in a segment is called segment sequence number. A segment carries an acknowledgement number as well. It is the expected sequence number of the next transmitted segment in reverse direction. Segments that carry data are put into a retransmission queue and a timer is started. If a segment is received acknowledging a segment sent before, the acknowledged segment will be removed from the retransmission queue. If the timer expires, the associated segment is considered lost and will be retransmitted. A received acknowledgement does not guarantee that the acknowledged data has been delivered directly to the application. It just guarantees that the data was delivered to the peers TCP layer and the application can consume it.

Flow control is accomplished by the "window". It is used to control the amount of data exchanged between the peers. With every acknowledgement a host sends its current window indicating the amount of bytes it is currently willing to accept. The window size is connected to the currently available buffer size, usually multiple times the maximum segment size (MSS). The MSS of each host is normally exchanged during connection establishment. This is necessary because TCP has no predefined limit for its payload size per packet.

Multiplexing between applications and the TCP Module is achieved by using port numbers. Every connection is identified by a pair of two port numbers, called source and destination. The source of a local application, is the destination of its peer. For example, a web browsers source is 24532 (randomly chosen) and its destination is a HTTP-server with port number 80. From the HTTP-server's point of view, the source port is 80 and the destination port is 24532. An IP-Address identifies a connection between two hosts, a port number between two applications, both combined identify a TCP connection uniquely.

Connection handling is necessary for the reliability and flow control mechanics mentioned above. They rely on initial exchange and maintenance of status information between both hosts. The status information includes sequence and acknowledgement numbers, window sizes, MSS, control flags and various TCP options. The information exchanged is stored in a data structure named transmission control block, short TCB. "When two processes wish to communicate, their TCP's must first establish a connection (initialize the status information on each side). When their communication is complete, the connection is terminated or closed to free the resources for other uses" [Pos81, p. 5]. Connection establishment is achieved by a handshake mechanism, specified later in this document.

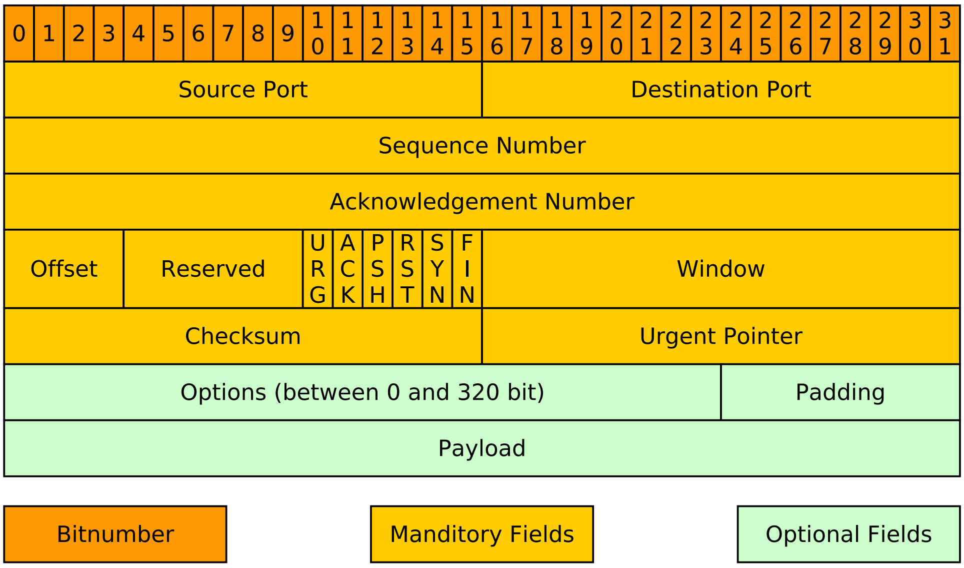

Figure 4: TCP header, see [Pos81, p. 15]

3.2 Header format

The TCP header stores control information needed for TCP operation. Its size ranges from 20 bytes, without any options to a maximum of 60 bytes limited by the 4 bit sized "offset"- field. This leads to a maximum of 40 bytes for options per segment. A TCP header enforces a 32-bit alignment, padding bytes are added after the option fields. Figure 4 shows the TCP header as defined in [Pos81, p. 15]. The header consist of:

Source Port and Destination Port: 16 bit each. The combination of both ports is used for multiplexing.

Sequence Number: 32 bit. If the SYN control bit is not set, the sequence number is the number of the first byte in this segments payload. If SYN bit is set, the sequence number is the senders initial sequence number (ISN).

Acknowledgment Number: 32 bit. If the ACK control bit is set, this field contains the value of the sequence number that the sender of the segment is expecting to receive next. Once a connection is established, an acknowledgement number is always sent.

Offset: 4 bit. Size of the TCP header expressed as the number of 32 bit words. It ranges from 5 (20 byte) to 15 (60 byte) depending on options in use.

Reserved: 6 bit. Reserved for future use. Must be zero in a TCP-implementation. The explicit congestion notification extension [RFB01] for example, uses the reserved field to extend the control bit field.

Control Bits: 6 bit. Contains TCP control flags.

URG: If URG is set, the Urgent Pointer field is significant.

ACK: If ACK is set, the Acknowledgment Number is significant, meaning this segment acknowledges earlier received bytes.

PSH: If PSH is set, the push function is used.

RST: If RST is set, the connection should be reset.

SYN: If SYN is set, the contents of the Sequence Number Field is the initial sequence number. This Flag is used for synchronization during connection establishment.

FIN: If FIN is set, the sender has nothing more to send. FIN indicates that the sender wants to close the connection.

Window: 16 bit. Contains the number of bytes the sender is currently willing to accept from the receiver. Originally limited to 65535 bytes, a larger receive window size can be communicated with the "Window Scaling" option [JBB92, p.8].

Checksum: 16 bit. Checksum to detect transmission errors. The calculation algorithm is specified in RFC 1071 [BBP88].

Urgent Pointer: 16 bit. Contains the offset from the sequence number until urgent data begins. This field is only interpreted if the URG control bit is set.

Options: variable (0 to 320 bit). The Option field can contain various options, built by the following scheme: The first byte is the type of an option, the second byte is the total length of an option in bytes, the following bytes are an options value. Some options consist only of the type field, they have no value. The "End of List"-option, for example consists only of a type-field with value 0. The MSS-option has a type field with value 2, a length field with value

4 and two bytes for the actual value of the maximum segment size. RFC 793 defines only the options End-of-List, No-Operation and Maximum Segment Size [Pos81, p.18]. TCP-Extensions like SACK [MMFR96] introduce additional options. The option-field must be aligned to a 32-bit boundary. If the option-field is not aligned, the remaining bytes must be filled with a padding composed of zeroes.

3.3 Transmission control block and sequence numbers

During TCP operation a connection state has to be stored. Both peers must store and maintain variables, organized in a data structure called transmission control block (TCB). The TCB contains variables to divide the sequence number space into different areas. In TCP a sequence number is assigned to every sent byte. A segments sequence number is associated with the first payload byte in a segment. The segments last payload byte sequence number is the sequence number plus payload size minus one. A whole packet can be acknowledged by sending an acknowledgement with an acknowledgement number bigger than the sequence number of the segments last payload byte. This mechanism enables simple duplicate detection, as well as the detection of missing packets. For each sent segment a retransmission timer is started. If a sent packet was not acknowledged before its retransmission timer expires, it would be assumed lost and the segment needs to be retransmitted.

Each host keeps track of used sequence numbers and received acknowledgement numbers in a connections TCB, the TCB defines a connections state directly. The stored variables inside the TCB divide the sequence number space into send sequence space and receive sequence space. Both sequence number spaces are further explained in this section.

The send sequence space state is maintained by the following variables: *SND_UNA* (Send Unacknowledged), *SND_NXT* (Send Next) and *SND_WND* (Send Window). Figure 5 shows the division of the send sequence space by those variables. Sequence numbers less or equal than *SND_UNA* have been successfully transmitted and acknowledged by the receivers side. Fully acknowledged segments must be removed from the retransmission queue. Sequence numbers less or equal than *SND_NXT* and greater than *SND_UNA* have been sent, but not yet acknowledged by the peer. It is unknown whether they were received or not, they have to remain in the retransmission queue. Sequence numbers greater than *SND_NXT* and smaller than *SND_UNA + SND_WND* can be sent to the peer. Sequence numbers bigger or equal than *SND_UNA + SND_WND* are outside the peers receive window, they are currently not permitted to be sent. For details, see [Pos81, p.55]. *SND_UNA* and *SND_WND* are updated with each

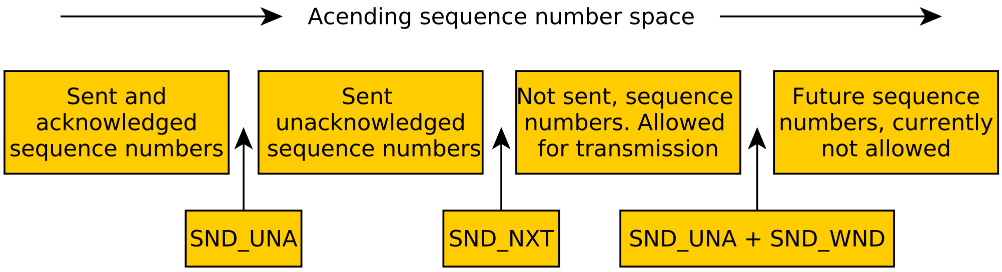
Acending sequence number space
Sent and acknowledged sequence numbers
Sent unacknowledged sequence numbers
Not sent, sequence numbers. Allowed for transmission
Future sequence numbers, currently not allowed
SND_UNA
SND_NXT
SND_UNA + SND_WND

Figure 5: Division of the send sequence space, see [Pos81, p.20]

received acknowledgement if the ACK-Flag in the TCP header is set. *SND_NXT* is advanced after each transmission by the size of the sent data.

The receive sequence space is the counterpart to the send sequence space. Each peer must maintain both number spaces. For receiver sequence space maintenance, the TCB stores the following variables: *RCV_NXT* (Receive Next) and *RCV_WND* (Receive Window). Figure 6 shows the receive sequence space division. Sequence numbers smaller than *RCV_NXT* have been received and acknowledged by the receiver. Sequence numbers between *RCV_NXT* and *RCV_NXT + RCV_WND - 1* are within the receive window and are processed further. After successful segment reception with a sequence number equal to *RCV_NXT*, *RCV_NXT* is advanced by the size of the received payload. The next segment sent acknowledges the sequence numbers up to the current value *RCV_NXT*. The acknowledgement is either piggybacked on the next segment that contains a payload or it is a pure ACK, meaning that this segment's only purpose is to acknowledge received data. If a received segments sequence number is within the receive window and the sequence number is not equal to *RCV_NXT*, a previous sent segment would probably be lost during transmission. As soon as the missing segment arrives, both segments can be acknowledged with a single acknowledgement. This is called cumulative ACK.

Additionally the transmission control block contains information about pointers to send and receive buffers, variables for urgent pointer handling, initial send and receive sequence numbers, pointers to the retransmission queue and to the current segment itself. Those contents should be mentioned, however, they are beyond scope of this thesis.

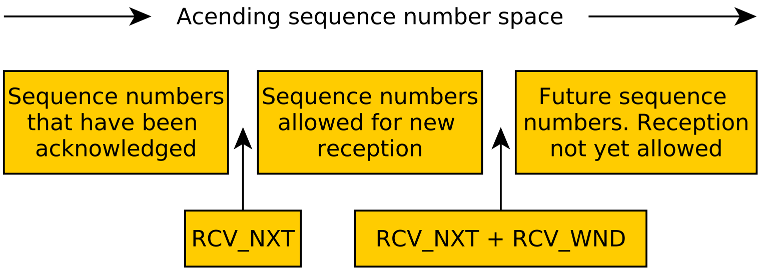
Acending sequence number space
Sequence numbers that have been acknowledged
Sequence numbers allowed for new reception
Future sequence numbers. Reception not yet allowed
RCV_NXT
RCV_NXT + RCV_WND

Figure 6: Division of the receive sequence space, see [Pos81, p.20]

3.4 TCP state machine

As a connection-oriented transport protocol, TCP needs to establish and terminate connections. The different states a TCP connection can pass through, are defined in the TCP finite state machine (FSM). Figure 7 shows a simplified version of this state machine. The simplified version illustrates only state changes in response to reactions, error conditions and error responses are omitted. A detailed description on the TCP FSM can be found in the "Event Processing" section of the TCP specification [Pos81, p.52].

Translations between the machine's states are triggered by three different kinds of events. A translation causing event can either be a function call from an application, a received packet from the peer or an expired timer. The FSM states are described below the descriptions originate from RFC793 [Pos81, p.21 - p.22]:

LISTEN - represents waiting for a connection request from any remote TCP and port.

SYN-SENT - represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED - represents waiting for a confirming connection request acknowledgement after having both received and sent a connection request.

ESTABLISHED - represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

FIN-WAIT-1 - represents waiting for a connection termination request from the remote TCP, or an acknowledgement of the connection termination request previously sent.

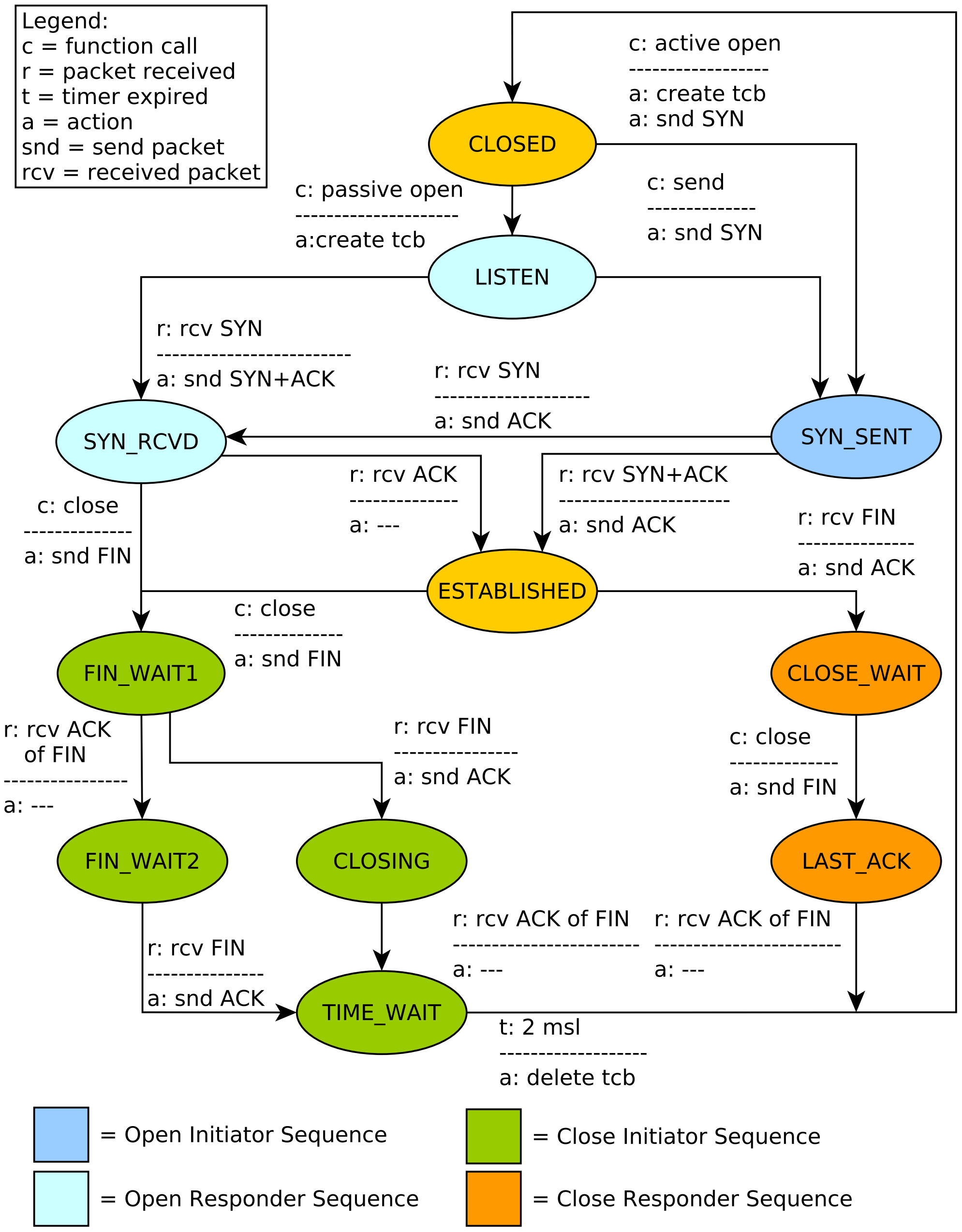

Figure 7: TCP state machine, see [Pos81, p.23]

FIN-WAIT-2 - represents waiting for a connection termination request from the remote TCP.

CLOSE-WAIT - represents waiting for a connection termination request from the local user.

CLOSING - represents waiting for a connection termination request acknowledgement from the remote TCP.

LAST-ACK - represents waiting for an acknowledgement of the connection termination request previously sent to the remote TCP (which includes an acknowledgement of its connection termination request).

TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgement of its connection termination request.

CLOSED - represents no connection state at all.

3.5 Connection establishment

First of all, for a reliable connection a session must be established. TCP connection establishment follows a 3-Way-Handshake. The handshake procedure ensures that both peers can contact each other, their sequence number spaces are synchronized and parameters necessary for TCP operation are exchanged.

Figure 8 shows the 3-Way-Handshake between a client and a server. An alternative simultaneous open procedure, where both peers perform an active open at the same time, is a special case of connection establishment and will not be further explained in this thesis.

To establish a connection, a server must perform a passive open, specifying a port number to wait for incoming connection requests. A client connects to the server by an active open call. The client allocates a TCB, generates an initial sequence number (ISN) and a source port number, before a SYN-packet is sent to the peer. The packet contains a set SYN-flag, the ISN of the client as sequence number and the client's local port as source port, the server's port as destination port. Additionally, the maximum segment size of the client should be transmitted too (via MSS-option).

After receiving the SYN-packet from the client, a server can accept the connection. The server needs to create a TCB, the TCB must be filled with the received SYN-packet information. The server needs to create its own ISN, in order to send it to the client. The server sends a packet to the peer, with SYN-and ACK-flags set. It contains the server's ISN as sequence number and the server's maximum segment size. Additionally this packet acknowledges the clients ISN by sending the clients ISN+1 as acknowledgement number.

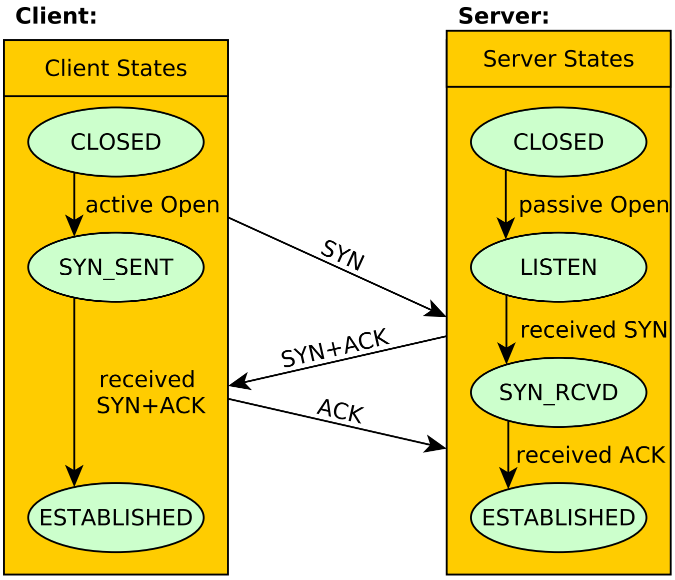

Figure 8: 3-Way-Handshake with state changes

After reception of the SYN+ACK-packet from the server, the client fills its TCB with the received information. From the client's point of view, the response from the server proves that the peer can be reached and is ready for a connection. The client acknowledges the server's ISN by sending a packet with an ACK-flag, its current sequence number and the server's ISN+1 as acknowledgement number. From the client's point of view the connection is established.

By receiving the acknowledgement from the client, the server assumes an established connection with the peer. The server's connection translates into the state "ESTABLISHED".

In cases of errors, non-verifiable sequence and acknowledgement numbers, absence of replies, either timers or transmissions of reset packets (RST-Flag set) are used to reset the host's state machines back to closed state. A more detailed description on the 3-Way-Handshakes sequence of events is in the TCP specification [Pos81, p.52].

3.6 Connection termination

Just as the TCP connection establishment, the connection termination follows a strict sequence as well. Both host applications need to close the connection independently by a close operation. The connection termination process is rather complex, due to the ability of TCP to send and

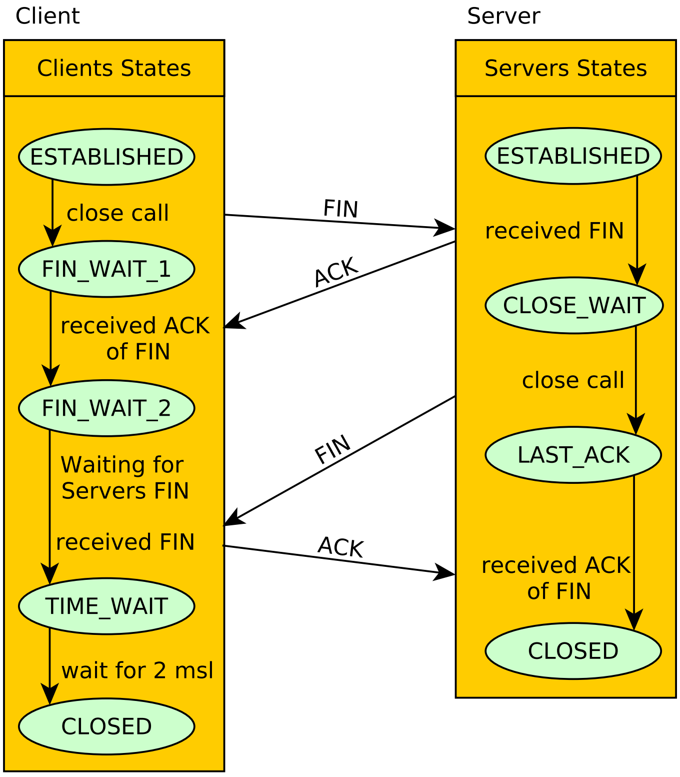

Figure 9: Connection termination sequence with state changes

receive data simultaneously. The connection termination differs from a simple reset. A normal connection termination closes a connection gracefully, a reset on the other hand is a forceful way to terminate a connection. A reset should only be used in case of errors or unexpected behavior.

A close operation indicates that the application has nothing more to send, but it is still able to receive and acknowledge data. Both peers must close their connection by sending a FIN-flag carrying packet and acknowledge the reception of the peers FIN-packet, to ensure every bit of data has been transmitted correctly. A host that acknowledges the last FIN-packet must wait two times the maximum segment lifetime (MSL) before translating into the closed state.

RFC 793 recommends a MSL of 2 minutes [Pos81, p.28]. By current network standards this is an unnecessarily long time. Current implementations should deploy a shorter MSL. This process ensures that the last ACK-packet was received by the peer. If not, the FIN-packet would be retransmitted before MSL timer expiration. With the translation into the CLOSED state, previously allocated resources, e.g. the TCB, are returned.

Figure 9 shows the flow sequence and state changes of a regular connection termination. The simultaneous connection termination as special case is beyond the scope of this thesis. In this example, the client ends the connection first, the procedure would be the same if the server starts the termination process.

3.7 Data transfer

In this section the TCP core service of data transmission is covered. For data transmission, the peers are required to establish a connection first. An application on the client's side sends data by calling the send function with the data to send. The client's TCP forms MSS sized segments for transmission. The client's TCP can send as much segments as long as they fit into the server's announced receive window. The amount of bytes that are currently allowed for transmission is the result of *SND_UNA + SND_WND - SND_NXT*. *SND_NXT* is advanced with every outgoing segment by the segment's payload length. Retransmitted packets have no effect on *SND_NXT*. Every outgoing payload or SYN/FIN-flag carrying segment is placed in the retransmission queue with its own timer. The timer expires shortly after the networks round trip time to ensure immediate retransmission.

With reception of a segment, the server's TCP layer checks if the carried sequence number is inside the propagated receive window. If the segment is acceptable and its sequence number matches *RCV_NXT*, the segment's payload is copied into the receive buffer and an acknowledgement segment is formed. The server application can now consume received data with the read function call. The acknowledgement segment can be piggybacked onto an outgoing segment or can be a pure acknowledgement. Either way the segment carries the value of *RCV_NXT* as acknowledgement number and the current receive window size. The receive window depends on currently available buffer sizes and is used for flow control to limit the amount of data the client can transmit to the server. The current window size it the amount of data, the server is prepared to accept.

With the reception of the acknowledgement, the client validates the transmitted sequence number. If the the received acknowledgement was piggybacked on a payload carrying segment, the payload is copied into the receive buffer and an acknowledgement is formed. By receiving a valid acknowledgement the value of *SND_UNA* is set to the received acknowledgement number and *SND_WND* is updated with the received window. Every segment inside the retransmission queue, that has been acknowledged, has to be removed from the queue. A packet will be acknowledged when the sequence number of the last payload byte is less than *SND_UNA* or the received acknowledgement number. Both have the same value at this point.

The time between the transmission of a segment and reception of the segment's acknowledgement is named round trip time (RTT). The RTT is the basis for the retransmission timeout (RTO) calculations. If the timer of a segment in the retransmit queue expires, the segment is considered lost. The segment needs to be retransmitted and its timer is restarted. The RTO calculation scheme from the TCP specification was updated in RFC 6298[PACS11] based on van Jacobson's work in [Jac88]. When a subsequent RTT measurement is made, a host must update the round trip time variance (σ_{RTT}) and the smoothed round trip time ($SRTT$):

$$\sigma_{RTT} = (1 - \beta) \cdot \sigma_{RTT} + \beta \cdot |SRTT - RTT|$$

$$SRTT = (1 - \alpha) \cdot SRTT + \alpha \cdot RTT$$

The factors α and β balance the influence of the new measurement on σ_{RTT} and $SRTT$. According to RFC 6298[PACS11], α should be $\frac{1}{8}$ and β should be $\frac{1}{4}$. Finally, the RTO is calculated with the following formula:

$$RTO = SRTT + k \cdot \sigma_{RTT}$$

The RTO is the smoothed round trip time plus the round trip time variance. The influence of σ_{RTT} is balanced by k (suggested value: 4).

The interaction between send and receive number spaces between two applications is visualized in figures 10 and 11. In this example a client requests a file from a server. The client's request measures 80 bytes in size, the requested file measures 300 bytes. The client application initiates the transmission by calling the send function with the 80 bytes long request. The server TCP receives this request and the server application accepts the data from the TCP by calling the read-function. The server's application processes the request and opens the requested file and calls the send-function with the requested file content. A reply is sent by the server's TCP acknowledging the client's request and containing the first 200 bytes of the requested file. The remaining 100 bytes are sent as a separate segment from the server, the client acknowledges the reception of each packet with an acknowledgement.

After the connection establishment phase the client's TCB contains the following variables: *SND_UNA = 1, SND_NXT = 1, SND_WND = 100, RCV_NXT = 1001, RCV_WND = 300.* The server's synchronized variables mirror the client TCB: *SND_UNA = 1001, SND_NXT = 1001, SND_WND = 300, RCV_NXT = 1, RCV_WND = 100.*

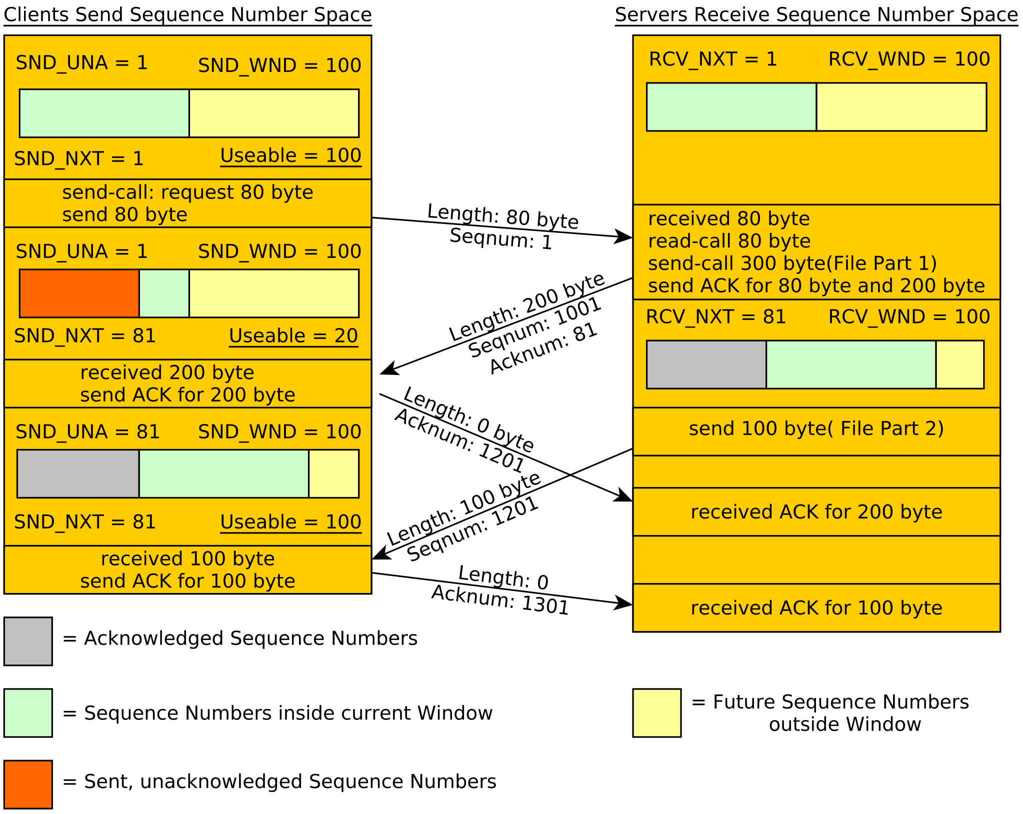

Figure 10: Data transfer example, clients send and servers receive sequence numbers

For simplicity this example assumes constant send- and receive window sizes. Another simplification is the absence of packet loss. A packet could be lost due to routing errors or network congestion.

Figure 10 shows the client's send sequence space and the server's receive sequence space. After sending the file request, the client's SND_NXT is advanced by the size of the sent payload (80 byte). This leaves 20 bytes left for further use inside the send window. The server's first reply contains 200 bytes payload and acknowledges the reception of the initial request. After receiving the acknowledgement, the client advances SND_UNA to the received acknowledgement number. In this scenario all data sent by the client has been acknowledged, the full send window of 100 bytes is available for further use.

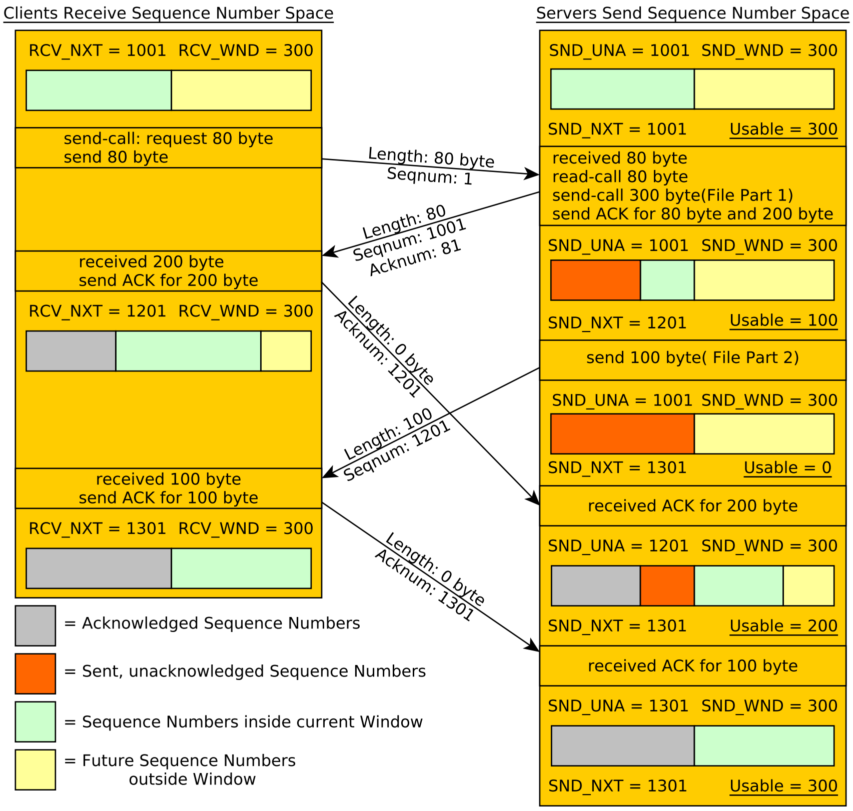
Clients Receive Sequence Number Space
Servers Send Sequence Number Space
RCV_NXT = 1001 RCV_WND = 300
send-call: request 80 byte
send 80 byte
received 200 byte
send ACK for 200 byte
RCV_NXT = 1201 RCV_WND = 300
received 100 byte
send ACK for 100 byte
RCV_NXT = 1301 RCV_WND = 300
= Acknowledged Sequence Numbers
= Sent, unacknowledged Sequence Numbers
= Sequence Numbers inside current Window
= Future Sequence Numbers outside Window
Length: 80 byte
Seqnum: 1
Length: 80
Seqnum: 1001
Acknum: 81
Length: 0 byte
Acknum: 1201
Length: 100
Seqnum: 1201
Length: 0 byte
Acknum: 1301
SND_UNA = 1001 SND_WND = 300
SND_NXT = 1001 Usable = 300
received 80 byte
read-call 80 byte
send-call 300 byte(File Part 1)
send ACK for 80 byte and 200 byte
SND_UNA = 1001 SND_WND = 300
SND_NXT = 1201 Usable = 100
send 100 byte(File Part 2)
SND_UNA = 1001 SND_WND = 300
SND_NXT = 1301 Usable = 0
received ACK for 200 byte
SND_UNA = 1201 SND_WND = 300
SND_NXT = 1301 Usable = 200
received ACK for 100 byte
SND_UNA = 1301 SND_WND = 300
SND_NXT = 1301 Usable = 300

Figure 11: Data transfer example, clients receive and servers send sequence numbers

Figure 11 displays the server's send sequence space and the client's receive sequence space. After reception of the request from the client, the server advances SND_NXT by 80 bytes and replies with a segment containing payload of 200 bytes in size, leaving 100 bytes left inside the server's send window. The requested file's second part covers 100 bytes. It fits exactly into the remaining send window. The server advances SND_NXT by 100 and sends a second segment with 100 bytes payload reducing the usable send window to zero. At this point the server cannot send more data as long as already sent data has not been acknowledged.

By reception of the first segment, the client advances RCV_NXT by the payload size of the received segment (200 bytes). The client sends an acknowledgement for the received 200 bytes. As soon as the server receives the acknowledgement, SND_UNA is advanced to the received acknowledgement number. This increases the usable send window to 200 bytes. After reception of the second segment, the client advances RCV_NXT further and sends an acknowledgement. The server receives the second acknowledgement, advances SND_UNA to the received acknowledgement number value, increasing the usable send window to 300 bytes. From the server's point of view all data has been acknowledged, the transmission was successful.

3.8 Window management

The transmitted window is the TCP mechanism for flow control. The window transmitted with every packet, is the source's receive window and will be the destination's send window after reception. It is assumed that the communicated window is related to the source's available receive buffer.

A large window encourages the transmission of a large amount of data. By announcing a large window, more segments can be sent before the sending TCP has to wait for incoming acknowledgments. The maximum segment size is the upper boundary for a segment's payload size. In modern TCP implementations a MSS is announced during connection establishment, the allocated receive buffer and the propagated windows are typically multiples of the transmitted MSS. The MSS size should be aligned with the networks MTU. The MTU can be determined dynamically with MTU path discovery [MDM96], otherwise the IPv6 default MTU (1280 byte) can be used. A common TCP MSS is 1220 bytes, based on the IPv6 MTU (1280 bytes) - the IPv6 header size (40 bytes) - the TCP header size (20 bytes without options).

A small window leads to the transmission of fewer segments, limiting the TCP throughput in general. The smaller the window, the fewer packets can be sent before the destination's receive window is exhausted, forcing the sender to wait for incoming acknowledgments and consumption of buffered data by the receiver.

TCP lacks a lower boundary for window size. There is no option to communicate a minimal segment size between the peers. However, TCP tries to fill segments before transmission. The receiver can stop transmission by announcing a zero window. This is called window closing. Even with a closed window, the receiving TCP must send an acknowledgement with the next expected sequence number and the currently available window size on incoming segments. The sending TCP must be able to transmit a packet with zero or one byte payload, even if the send window is zero. This so-called probe segment, must be transmitted on a regular basis until the receiver announces a non-zero window. This mechanism is essential to guarantee that the window re-opening is reported reliably.

The size of the sliding window has a serious influence on TCP performance. An increase in TCP performance is achieved by avoiding the transmission of small segments. Basic TCP has a tendency to announce smaller windows in situations where the receiver is under heavy load. This leads to more transmitted packets, decreasing the receiver- and network performance further. This phenomenon is called "Silly Window Syndrome". Additional information on window management, the cause and effects of the silly window syndrome can be found in RFC 1122 [Bra89, p.89-90].

The silly window syndrome can be avoided by not announcing small windows on the receiver's side and not sending small segments on the sender's side. The receiver should not allow the propagation of a small window. Instead the window should be closed entirely until a reasonable window can be propagated. The receive window can be re-opened as soon as the receive buffer is able to store at least one MSS sized segment.

On the other hand, the sender should avoid sending small segments. An algorithm addressing this specific problem is Nagle's Algorithm specified in RFC 896 [Nag84]. Nagle's Algorithm tries to send MSS-sized packets, if possible. Its core concept is simple:

1. If the amount of outgoing data and the receive window are bigger or equal than the receivers MSS, send a segment with MSS sized payload. This always leads to reasonable sized segments.

2. If data has been sent and is still unacknowledged, outgoing data will be stored and accumulated, until all previously transmitted data has been acknowledged or the accumulated data is bigger or equal to the receiver's MSS. This accumulation enables TCP to send fewer, larger segments instead of more smaller segments. Additionally it adjusts the rate of outgoing segments to network load conditions by awaiting the acknowledgments.

TCP flow control and error handling are important issues but the basic TCP standard is very vague in this topics. The important topic of network congestion handling is not covered in TCP. These subjects have been addressed by various TCP-extensions. The extensions "Selective Acknowledgment Options" and congestion control mechanisms are covered in the next chapter of this thesis.

4 TCP extensions and the Internet of Things

Since the basic version of TCP was introduced in 1981, TCP has been extended multiple times to cope with the excessive growth of computer networks and paradigm shifts in network technology. With the advent of the wireless embedded Internet, new types of computer networks are on the rise. Those networks differ from established wired networks considerably. Nodes in the IoT are often battery powered with low computing-power and they are connected spontaneously over unreliable wireless network interfaces.

In this section, selective acknowledgment options and congestion control mechanisms are presented and evaluated for their suitability for the Internet of Things.

4.1 Selective acknowledgment options

The loss of a single packet in TCP often leads to unnecessary retransmissions. If a single segment inside a series of segments is lost, received segments can only be acknowledged up to the sequence number of the missing packet. Received unacknowledged packets, cannot be acknowledged until the missing segment is retransmitted and delivered successfully. These received unacknowledged segments are often retransmitted unnecessarily, because their retransmit-timers expire, due to the delay caused by the retransmission of the single lost segment. This behavior becomes more problematic with decreasing reliability of the involved link-layer technologies, leading to more packet losses and unnecessary retransmissions, decreasing performance further. Basic TCP has no way to communicate that a segment was lost and the following segments were received and do not need to be retransmitted.

The selective acknowledgement options, in short SACK, address this specific problem. SACK is standardized in RFC 2018 [MMFR96]. The underlying idea is to send an option with every acknowledgement, specifying the ranges of received sequence numbers that cannot be acknowledged currently. With this information a sender can deduce which segments need to be retransmitted, and which segments were received but cannot be acknowledged yet. The received segments remain at the sender's retransmit-queue until they have been acknowledged cumulatively. However, they are marked as received, stopping their retransmission-timers until the lost segment has been retransmitted. With the retransmission of the lost segment,

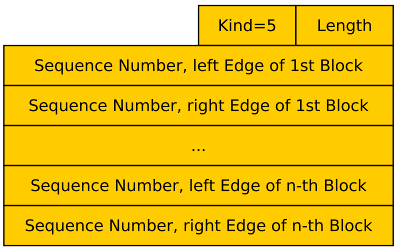
Kind=5
Length
Sequence Number, left Edge of 1st Block
Sequence Number, right Edge of 1st Block
...
Sequence Number, left Edge of n-th Block
Sequence Number, right Edge of n-th Block

Figure 12: SACK option format, see [MMFR96, p.3]

the mark on the received segments should be removed and their retransmission-timers should continue.

Selective acknowledgments introduce two new options into TCP. The first option is called SACK-permitted. This option is sent during the 3-Way-Handshake. The SACK-permitted option is only allowed in packets where the SYN-Flag is set. This option is used to communicate that SACK can be used after successful connection establishment. The option itself consists of two fields, the Kind-field with value 4 and the Length-field with value 2. For SACK usage, both peers must send a SACK-permit on connection establishment.

The second option is the actual SACK option. A SACK option exists only in segments with an ACK-Flag set. It enables the receiver to communicate multiple received sequence number ranges. Each coherent received sequence number range starts with the first received sequence number called left edge and ends with the last sequence number named right edge, in the received number space. Each sequence number consumes 4 bytes, the kind and field option occupy 1 byte each, leading to a total memory consumption of $8 \cdot n + 2$ bytes in the TCP option field. The options field can carry up to 40 bytes leading to a maximum of four distinct sequence number ranges. Figure 12 shows the SACK option format as defined in RFC 2018.

Let us clarify SACK operation with three examples. We assume a scenario with four packets sent in a burst, each segment is carrying a payload of 100 bytes. The first segment carries the sequence number 100.

In the first example the third and fourth packets are lost. The reception of segment one and two leads to an acknowledgement, acknowledging the reception of both segments. The sent acknowledgement segment contains 300 as acknowledgement number. It carries no SACK option, because no segment has been received out of order.

For the second example the loss of the first two packets is assumed, packet three and four were received. The reception of packet three and four leads to an acknowledgement although there is no new data that can be acknowledged. This segment contains 100 as acknowledgement number and a SACK option with one received coherent number space, stretching over the two received packets. The left edge is 300 (sequence number of the third segment) and the right edge is 500 (sequence number of the fourth segment + segment's size).

In the last example we assume a loss of the first and the third packet. In this case, the received segments trigger the formation of an acknowledgement. This acknowledgement contains the acknowledgment number 100, because no data can be acknowledged. Additionally this ACK contains a SACK option as well. The SACK option contains two separate number spaces. One number space spanning over the contents of the second segment with the left edge of 200 and the right edge of 300. The second number space spans over the fourth segment's number space, the left edge is 400, the right edge is 500.

Selective acknowledgment options in the IoT could be adequate countermeasures to the inherent unreliability of wireless technologies like IEEE 802.15.4 based standards. By using SACK, lost packets can be identified precisely, preventing already received packets to be retransmitted needlessly. This minimizes the amount of segments that need to be transmitted over the network. Every unsent packet does not need to be forwarded by other nodes either. With the mesh-networking context in mind, SACK can help to reduce the power-consumption of every node that routes a received segment and of both end nodes.

The most important disadvantage of SACK for constrained nodes is an increased memory requirement on the receiver's side. In a simple memory efficient TCP implementation, a receiving node could restrict packet reception to one segment via its announced window, minimizing memory footprint and leading to poor performance in general. The usage of SACK implies that the receiver is able to receive multiple segments and store received out-of-order segments, increasing packet and receive buffers. Those out-of-order segments can be acknowledged accumulatively, after the successful reception of the missing segments. The amount of additional memory needed to store out-of-order segments, depends on the receivers communicated maximum segment size. As discussed in the window management section, the receive window size should be MSS multiple times. A SACK implementation should be able to store as much data as advertised in the propagated window.

Additionally in the IoT, small receive window sizes of only one or two segments are common. In such scenarios SACK deployment is nearly useless because the reception of out-of-order segments is unlikely. With the low usefulness of SACK in the IoT, SACK is rarely deployed, and SACK needs to be implemented on both hosts for SACK operations.

Another disadvantage of SACK is the inflation of the TCP header information. A TCP header without any options measures 20 bytes, with SACK the header measures up to 54 bytes ($20 + 8 \cdot 4\ Segments + 2 = 54$) in the worst case. This decreases the ratio between header size and payload considerably, especially in IoT scenarios where small payloads are common.

In a nutshell, using SACK seems to be not useful on restricted nodes. Efficient SACK operation leads to increased memory requirements, that might be too heavy for constrained nodes.

4.2 Congestion control

The main cause for packet loss in networks with wired links is network congestion. Routers drop packets in case of overload, leading to missing segments. By dropping packets, the retransmit mechanism sends additional segments increasing the congestion problem further. The detection of network congestion enables TCP to adjust the transmission rate to the current network capability. In modern TCP implementations, the detection and handling of network congestion is achieved by four intertwined algorithms named slow start (4.2.1), congestion avoidance (4.2.2), fast retransmit (4.2.3) and fast recovery (4.2.4). The algorithms were originally introduced by Van Jacobson in RFC 2001 [Ste97], the RFC was updated by RFC 5681 [APB09].

4.2.1 Slow start

The original TCP standard covers window management only as a flow control mechanism, there are no restrictions on initial transmission rates. Each host is allowed to send data as fast as possible after the connection establishment. The slow start algorithm increases the transmission rate exponentially by defining an upper bound for the amount of data to transmit currently. The algorithm maintains a congestion window at the TCB, the congestion window is initialized with a value based on the received MSS. Each time a sent segment has been acknowledged successfully, the congestion window is increased by the minimum of the receiver's MSS and the number of acknowledged bytes. The sender is only allowed to send data up to the minimum of the propagated window and the congestion window. For the following example of slow start, we assume a MSS of 100 bytes, the check against the receive window size is omitted for simplicity.

The sender begins by sending 100 bytes sized segment and waits for the reception of an acknowledgement for this segment. On reception of the acknowledgement, the congestion window is increased by MSS, measuring 200 bytes in total. The sender is now allowed to send a burst of two 100 bytes sized segments. With the acknowledgement reception of each segment, the congestion window has been increased by 100 bytes each time. Now the congestion window allows a transmission of 400 bytes in a packet burst.

4.2.2 Congestion avoidance

By interpreting packet loss as a sign of network congestion, the expiration of retransmit-timers or the reception of multiple acknowledgments for the same segment (called duplicate ACK), are signs of packet loss and therefore of network congestion. The congestion avoidance algorithm offers a linear increase in transmission rate in contrast to slow starts exponential increase. Congestion avoidance introduces a "slow start threshold"-variable into the TCB. This variable stores a threshold value to differentiate between usage of slow start and congestion avoidance, balancing the data transfer growth.

In case of network congestion, detected by expiration of a retransmit-timer, the threshold is set to the maximum between the amount of data currently traveling through the network divided by two, and the sender's MSS times two [APB09, p.7]. Additionally the congestion window is set to the MSS value, ensuring usage of slow start for transmission rate increase. On reception of duplicate acknowledgments the network might be congested. However, data flow between peers is still possible, the current congestion window will not be reduced.

With every incoming acknowledgement, the congestion window is compared to the current threshold value. If the congestion window is less than the threshold, the connection uses the slow start to increase the data flow fast. Otherwise, the congestion window is increased by $\frac{MSS^2}{congestion\ window}$. This formula offers an adaptive, linear growth in transmission rate to approach the network's available bandwidth while avoiding network congestion.

4.2.3 Fast retransmit

Network congestion is indicated by the expiration of a retransmit timer or the reception of duplicate acknowledgments. The fast retransmit algorithm is the major source of these duplicate acknowledgments. In case of packet loss caused by network congestion, the receiver might receive the segments following the dropped segment. These segments are received as out-of-order segments. On reception of an out-of-order segment, the receiver must send an acknowledgement immediately. This acknowledgement cannot acknowledge new data, therefore this is a duplicate ACK for already acknowledged data.

Although from the sender's perspective, the reception of a duplicate ACK is a strong indicator of packet loss. However, there is still the possibility that other network problems caused the duplicate acknowledgments. To be sure that a specific segment was dropped and needs to be retransmitted, at least three duplicate ACKs with the same acknowledgement number must be received by the sender. With reception of the third duplicate acknowledgement, the sender retransmits the lost segment resetting the segment's retransmit-timer. Fast retransmit is only suitable to recover from single losses in a burst of packets.

4.2.4 Fast recovery

The fast recovery algorithm ensures that after a retransmission caused by the fast retransmit algorithm, congestion avoidance is performed instead of slow start. By receiving multiple duplicate ACKs, the peer must have performed fast retransmit. Fast retransmit is only triggered on receiving segments, therefore data flow between the peers is still possible despite current network congestion. By performing slow start, the transmission rate would drop immediately because slow start restarts with one MSS-sized segment. As long as there is still data flowing, congestion avoidance can be performed, avoiding the transmission rate drop on slow starts initialization.

The fast recovery algorithm is implemented as follows. With reception of the third duplicate acknowledgement, the slow start threshold is set to maximum between the amount of data currently traveling through the network divided by two, and the senders MSS times two. After retransmission of the lost segment, the congestion window is increased by MSS times three plus the threshold value. This inflates the congestion window by the number of segments that caused the duplicate acknowledgments. With reception of each additional duplicate ACK, the congestion window is increased by MSS to reflect that a segment has left the network.

With the first acknowledgment that acknowledges new data, the congestion window value is set to threshold size, leaving the connection always in congestion avoidance mode.

4.2.5 Congestion control in the IoT

The four intertwined congestion control algorithms can be useful in an IoT context, although the basic assumption behind these algorithms does not hold. In wireless networks packet loss is not only caused by network congestion. Packets may also be corrupted due to distorted signals, for example. If the lost packets are part of a series of transmitted packets, fast retransmit can shorten the delay for the lost packet to be retransmitted considerably.

Problematic for congestion control are small window sizes, common in memory saving TCP implementations. The congestion window specifies the maximum amount of data that can be transmitted as a burst of packets. Before packaging data to send, the amount of data that should be transmitted is calculated as the minimum of congestion window and send window. Especially in the Internet of Things small send windows are common. The congestion control algorithms change the congestion window, but if the receiver announces small window sizes, the send window is usually less than the congestion window. The sender is able to send the amount of data up to the send window, which is less than the congestion window. In such a scenario, it makes no difference whether the sender performs slow start or congestion avoidance.

Furthermore, some congestion control algorithms lead to increased memory requirements to function properly. The four algorithms itself are memory saving, but fast recovery and fast retransmit rely upon the reception of at least three duplicate acknowledgments. To send three duplicate acknowledgments, three out-of-order segments must be received and stored, as well as the missing segment on reception. With a common maximum segment size of 1220, oriented on the IPv6 MTU, this leads to minimal receive buffer requirements of nearly 5 kilobytes.

On the other hand, a modified fast retransmit algorithm could increase transmission rates in IoT scenarios. We assume a constrained node that is able so send a burst of two segments. On these nodes slow start, congestion avoidance and fast recovery are useless. Normal fast retransmit would not be triggered, because only one out-of-order segment can be received by the peer. A simple solution is the transmission of three acknowledgments on reception of an out-of-order segment. This modification causes fast retransmit to send the missing segment immediately, but in cases of packet reordering unnecessary retransmissions occur.

In summary, fast retransmit can increase TCP performance in IoT scenarios. The other algorithms require TCP implementations that are able to send bursts of multiple segments to have any significant effect. Such nodes are not the norm in IoT networks due to the increased memory requirements caused by segment buffering.

5 Related work on embedded TCP/IP stacks

This section covers the characteristics of two network stacks designed for embedded devices called µIP and lwIP with respect to their TCP implementations.

5.1 µIP

Micro IP[13] (µIP) is a popular TCP/IP stack developed by Adam Dunkels. The µIP stack was initialized by the "Networked Embedded Systems Group" of the Swedish Institute of Computer Science (SICS) and is licensed under the BSD license. The target platforms for µIP are 8 and 16-bit microcontrollers with at least 5 kB RAM and approximately 30 kB ROM. Micro IP is deployed with Contiki and the Arduino Ethernet shield, it can be used as a stand alone version as well. Originally, µIP offered only IPv4 support, a collaboration of Cisco, Atmel and SICS extended µIP to be fully IPv6 compliant.

The TCP/IP implementation supplies a global buffer to store incoming packets. The buffer size depends on a configurable maximum packet size. The Micro IP packet buffer can store a single packet at a time. An incoming packet needs to be processed by an application before the next packet arrives or the following packet is discarded on arrival. Micro IP offers a different network API than the de facto standard BSD sockets, increasing the effort to port existing network applications. Additionally, µIP relies on an event driven application design. In case of incoming packets an event is sent to the application. The application is expected to process received data immediately instead of buffering data until it is consumed. Protocols like UDP can be removed from µIP via Macros in the configuration file, reducing code size significantly.

The TCP implementation of µIP features a few specific characteristics, developers need to keep in mind. For example µIP omits a retransmission buffer. To reduce the amount of memory required for µIP usage, outgoing data is not buffered. In case of a retransmission, an application must be able to recreate the exact packet that was transmitted before. From µIP's point of view there is no difference between a first time transmission and a retransmit. Therefore the

[13] `https://github.com/adamdunkels/uip`

network stack and an application are not clearly separated, applications need to implement callback functions to work properly.

The window management strategy of µIP is simple. The predefined MSS equals the receive buffer size. The MSS is announced with the MSS option on connection establishment. The announced window is not changed in any way. There is no window closing or reopening mechanism. It is just assumed, that an incoming packet is processed before the next arrives, eliminating the need for window maintenance.

The unmodified version of µIP sends only one packet at a time and waits for an acknowledgement, until the next packet can be sent. A known issue with this approach is the delayed ACK functionality deployed by most full featured TCP implementations. Delayed ACK reduces the amount of sent acknowledgments by waiting for multiple incoming packets. As soon as multiple packets were received, a single acknowledgement is sent to acknowledge them collectively. By sending a single packet at a time µIP performance suffers because either a retransmission timeout occurs or the acknowledgment from the peer is delayed.

5.2 lwIP

Lightweight IP[14] (lwIP) is another TCP/IP stack with a focus on embedded devices. Like µIP, lwIP has been initiated by Adam Dunkels at the SICS. Currently it is developed by an international team as an open source project. Detailed information on the architecture of lwIP can be found in [Dun01]. Lightweight IP targets 8- and 16-bit microcontroller platforms with at least 10 kB RAM and approximately 40 kB ROM. The lwIP TCP implementation focuses more on feature completeness, transmission performance and portability than µIP, leading to increased memory requirements. Compared to µIP, lwIP decouples applications from the network stack. Developers do not have to implement several callback functions to use lwIP. If lwIP is deployed with an operating system like FreeRTOS, applications do not have to comply to an event driven design anymore. Additionally, lwIP offers multiple APIs to access the network stack including a BSD socket API as a wrapper for its netconn API. This improves portability between platforms. The usage of netconn and BSD socket API requires RTOS functionality, the raw API can be used in an application without an operating system, requiring the application to implement several callback functions.

The TCP implementation of lwIP features basic TCP functionality as well as TCP extensions suitable for embedded devices. The lwIP implementation supports transmission of multiple segments before expecting an acknowledgement, buffering of incoming and outgoing data

[14] http://savannah.nongnu.org/projects/lwip/

and handling of received out-of-bounds segments. Delayed acknowledgment functionality and the congestion control algorithm's slow start, congestion avoidance, fast retransmit and fast recovery are implemented. The silly window syndrome is avoided by announcing a window of at least MSS size, outgoing data is queued until a reasonable window size is announced from the peer. The lightweight IP stack is not feature complete, urgent pointer functionality, SACK and window scale options are not implemented in lwIP.

The next chapter covers the concepts and design goals behind the RIOT-OS gnrc TCP implementation. The design and implementation are inspired by the design of µIP and lwIP.

6 Concepts of gnrc TCP

This chapter covers the specific design goals for the gnrc TCP implementation and the software architecture in general. The design goals arise from the specific context RIOT is used in. Each design goal is explained in detail as well as methods to accomplish them.

6.1 Design goals for an embedded TCP implementation

Memory efficiency over network throughput: On constrained devices the amount of available memory is the most limited resource. Normally IoT applications are not designed to transmit large amounts of data. Therefore memory efficiency is favored to high transmission rates.

Static memory allocation only: To preserve the real-time capabilities and to enforce a deterministic memory consumption during runtime, dynamic memory allocation is not allowed in the RIOT core facilities. As a part of the generic network stack, TCP is restricted to use static memory allocation exclusively.

Optional TCP support: The modular design of the generic network stack enables users to customize the entire network stack at compilation time. Modules that are not used by an application, should not be compiled and linked into the binary to minimize application size. Following this philosophy, the TCP support must be optional and should only be included if requested.

Optimized feature set: TCP, especially with its TCP extensions, is a complex and resource demanding protocol. The gnrc TCP implementation provides an optimized feature set optimized for the IoT. This reduces code complexity and memory consumption.

Interconnectivity with other TCP implementations: The purpose of TCP is to supply reliable exchange of data between applications regardless of the underlying operating system. Gnrc TCP must be standard compliant to ensure interoperability between nodes running RIOT and nodes running other operating systems. The new RIOT TCP implementation must be able to handle connections with TCP implementations that offer a much larger feature set.

Integration into the new RIOT socket API: The rewrite of the RIOT network stack introduced a new socket API named "conn"[15]. By being an essential part of the network Stack, TCP has to be integrated into the RIOT socket API.

6.2 Software architectural overview

The functional TCP specification in RFC 793 [Pos81] describes the TCP FSM behavior in the event processing section starting from page 52 to 79. The different events which can occur on a TCP connection are grouped into three categories.

1. Function calls from user space applications.
2. Reception of a packet from the peer.
3. Expiration of a timer associated with a timer event.

Figure 13 contains an architectural overview of the gnrc TCP implementation. The central module of gnrc TCP is the finite state machine (FSM) of a connection, implemented as a set of c-functions. The FSM controls a connections behavior by responding to specific events. A FSM changing event can be triggered by a user calling a function from the gnrc TCP API for example. Other sources for FSM changing events are expired timers or the reception of packets. Figure 13 shows modules that can access a connections FSM and message exchange between the FSM and these modules.

Events that correspond with function calls from user space are triggered by an application, calling functions from the gnrc TCP API. The caller blocks until the called function has been finished. A function, called to open a connection would block until the connection has been established or the timer for the user space function calls has expired. In the latter case, the connection will be closed. To notify the waiting application thread, the FSM sends messages to the blocked thread in case of an important event. A thread calling the send function e.g. would be notified that send data has been acknowledged. On notification reception the calling thread can exit the called function.

The TCP connection handling thread is a part of the gnrc network stack. It passes packets down the network stack, to lower layers and receives incoming TCP packets from the lower

[15] `http://riot-os.org/api/group__net__conn__tcp.html`

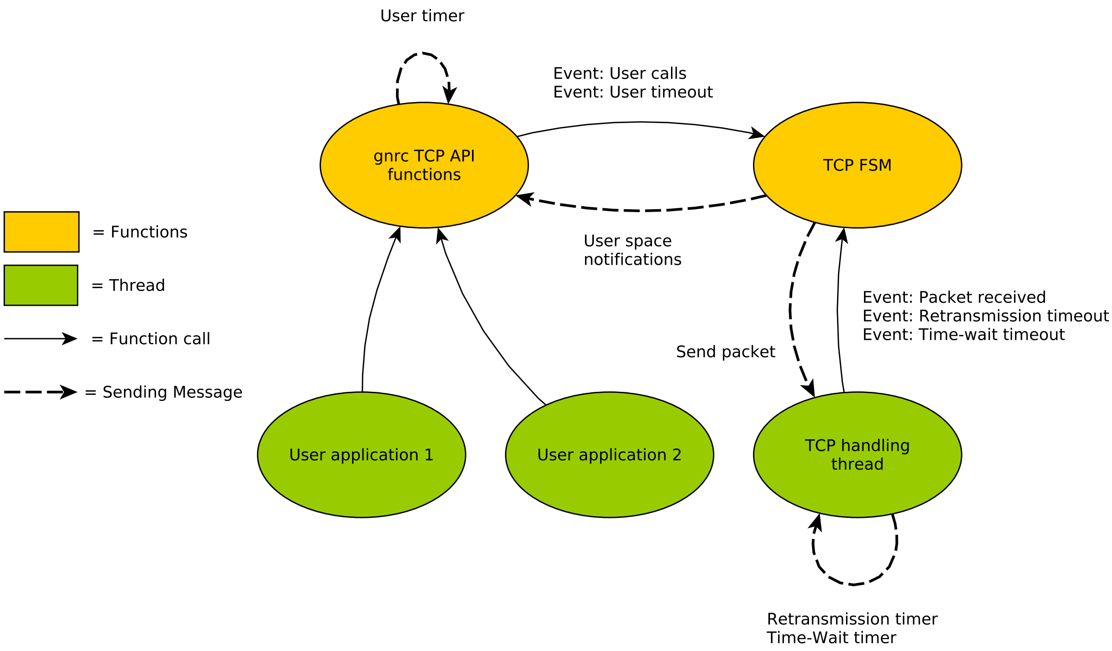

Figure 13: Architecture of the gnrc TCP implementation

layers. On packet reception, this thread performs verification and de-multiplexing by searching for the transmission control block (TCB) associated with the received packet. As soon as a fitting TCB has been found, the TCP handling thread calls this connection's FSM with a received packet event.

The transmission of a segment can a reaction to an occuring event during TCP operation. In this case a packet is allocated inside the packet buffer and a message is sent from the FSM to the TCP handling thread containing a pointer to the allocated packet. This message is processed and passed down the network stack for further processing and transmission.

The third category of TCP events are based on the expiration of specific timers. By calling a TCP API function, a timer is started. If this timer expires, it is assumed that the TCP connection is interrupted, triggering the according FSM event to close the connection. This timeout indicates that a connection has been aborted, because the peer is not responding anymore. It prevents the calling thread to be blocked forever in case of connection loss. A retransmission timer is started when a payload or SYN/FIN-flag carrying segment is sent to the peer. On timer expiration, a message is sent to the TCP handling thread triggering the FSM event to send the segment again. The time-wait timer expiration is a normal part of TCP connection termination. As soon as this timer expires, the connection is considered closed. Retransmission and time-wait timer can be combined into a single timer. Timer related events occur from threads in user space and from the TCP handling thread.

To summarize the software architecture section, FSM changing events can occur concurrently from application threads and from the TCP handling thread. To prevent undefined behavior and race conditions, the entire FSM is a critical section. Every manipulation on a connection's state must occur inside the FSM exclusively. For synchronization purposes each connection stores a mutex inside its TCB. The mutex of a connection must be acquired on FSM entry and released on exit of the FSM function.

6.3 Reducing memory requirements

The memory consumption of TCP depends mostly on the announced receive window. TCP performance can be improved by announcing a large receive window allowing the peer to send more segments before the window has to be closed. The increased transmission rate comes at the price of larger memory requirements. The receiving node must be able to store the received data until it is consumed by the application. A larger window leads to larger receive buffers.

The gnrc network stack supports only IPv6 as networking protocol. IPv6 requires a MTU of 1280 byte. Subtracting the fixed TCP and IPv6 header sizes from the MTU, 1220 bytes are left for the maximum segment size. By announcing a MSS of 1220 bytes on connection establishment, TCP should never receive a payload larger than 1220 bytes per segment.

A full-featured TCP implementation would announce a window, multiple times the MSS in size. This allows multiple connection related packets to travel through the network before the window is full. The gnrc TCP implementation announces configurable receive window size, the default configuration announces a receive window of MSS size (1220 byte). This limits throughput to a single, MSS sized packet traversing the network before an acknowledgement is expected. This window size limitation reduces the size of the necessary receive buffer to a maximum segment size of 1220 bytes per connection.

Another way to reduce gnrc TCP memory requirements is omitting the send buffer. TCP normally supplies a send buffer per connection. This buffer is filled with data an application wants to send. After copying the data into the send buffer, the send function can return and the TCP handling subsystem can transmit the copied data later. The entire send buffer can be omitted by direct transmission. Instead of copying data for later transmission, it can be directly segmentized and sent from data containing buffer the user supplied. Omitting the send buffer has a drawback. The connection is prune to transmission errors. On connection abort, a TCP implementation with a send buffer can presume normal operation. A TCP implementation without send buffer blocks the application thread until the user function call timeout occurs.

The gnrc TCP sending strategy is simple. Only one packet is transmitted at a time. The next segment will be sent as soon as the previously sent packet has been fully acknowledged. This strategy reduces the memory requirements significantly. By sending only one segment at a time, the segment needs to be stored in the packet buffer and in the retransmission queue. By implementing deduplication in the central packet buffer, the memory requirements are reduced further. Each object stored in the packet buffer, keeps track of its current users. The user counter is decremented with every thread releasing the packet. As soon as the user counter reaches zero, the packet is removed from the buffer. After processing and transmission of a packet, the user counter for a packet is normally zero, the packet is removed from the packet buffer. Packets containing payload or control flags like SYN and FIN must be persistent for a possible retransmission. By adding an additional retransmission user, these packets are persistent inside the packet buffer after their initial transmission. The retransmission user releases fully acknowledged packets on acknowledgement reception. After releasing the packet the user counter reaches zero and the packet is removed from the packet buffer. This simple mechanism removes the need to copy transmitted segments in a separate retransmission queue. For the retransmission mechanism a timer structure and a pointer to the segment inside the packet buffer are needed. Both are stored inside the TCB of each connection.

6.4 Static memory allocation

As a operating system with real-time capabilities, the RIOT core facilities must use static memory allocation exclusively. This is a prerequisite to hold real-time guarantees, and a general rule the gnrc TCP implementation must comply to. Additionally, by forbidding dynamic memory allocation the memory consumption behaves deterministic during runtime.

The data structures holding a TCP connection state are normally stored in the TCB, which is part of a socket. In the BSD socket implementation the "socket"-function creates a socket and returns a file descriptor to the newly created socket. This mechanism relies on dynamic memory allocation because sockets are allocated at runtime.

RIOT currently has no concept of file descriptors and dynamic memory allocation is forbidden for RIOT core facilities. This makes an exact reimplementation of the BSD socket API hard to achieve. Additionally, the TCP handling thread must be able to search all existing transmission control blocks for de-multiplexing purposes. Central access to all active transmission control blocks must be provided.

To address this problem a linked list is used. Each TCB contains a pointer to the next TCB. An initialization function inserting a new TCB into the linked list is supplied. After list assembly, the TCP handling thread can search for a specific TCB by iterating over the list. By using a linked list to connect transmission control blocks, allocated on thread's stacks, dynamic memory allocation can be avoided entirely. The TCB data structures are allocated either in an application's static section or in a thread's stack. The location a TCB is allocated must be valid as long as the connections exists. Before connection establishment the TCB has to be initialized, after connection closing the TCB must be destroyed with the appropriate function supplied by the gnrc TCP API. The linked TCB list could span across multiple thread's stacks and can only be assembled because RIOT uses a single memory layout instead of protecting the thread stack against access outside of a thread's context.

Each gnrc TCP API function expects a pointer to the TCB, to specify the connection the operation should applied on. This is similar to the BSD socket interface, but BSD sockets expect a file descriptor instead of a TCB pointer.

6.5 Optional TCP support

The most important characteristic of the gnrc network stack is its modularity. As part of the build process each application specifies modules that are used by the application. Modules that are not used should not be compiled and are not started on system initialization.

The TCP module must be optional. Applications using TCP must specify TCP usage in the projects make file by adding "USEMODULE += gnrc_tcp". This triggers auto initialization of the TCP handling thread during the RIOT initialization phase as well as the compilation of all TCP related functions.

6.6 Optimized feature set

TCP is a complex protocol which has been extended many times. Most extensions improve data throughput at the cost of memory consumption. However, the most important design goal of this TCP implementation is the reduction of memory requirements. The features gnrc TCP provides must be analyzed carefully, reducing the feature set to the bare minimum necessary for TCP operation. Options like SACK are not designed for an IoT context, not supporting them will reduce code complexity considerably.

In the gnrc TCP implementation the only implemented option is MSS to limit the segment's size a RIOT node can receive. Options like push and urgent and the fast retransmit extension are omitted, too. A future version of TCP might implement them. The urgent function allows prioritized data inside a TCP stream, allowing the communication of prioritized data without the need to open a second connection. With the current transmission strategy, implementing fast retransmit is pointless. Fast retransmit relies on the transmission of at least four segments at a time, the first version of gnrc TCP sends only one segment before expecting an acknowledgement.

6.7 Interconnectivity with other TCP implementations

The purpose of transport protocols is the interconnection of applications across operating systems. The gnrc TCP implementation must be compliant with the TCP implementations of other operating systems, testing against other operating systems TCP implementations must be an integral part of the gnrc TCP testing procedure. Although gnrc TCP provides only an optimized feature set, it must be able to exchange data with full-featured TCP implementations. Options unknown to gnrc TCP must not lead to undefined behavior.

6.8 Integration into the new RIOT socket API

With the generic network stack a new API is introduced into RIOT. The "application connection" API (conn) features a unified interface to the gnrc network stack similar to BSD sockets. In the current version, all TCP specific functionality is defined as unimplemented function prototypes. Like UDP, the gnrc TCP implementation must comply to this predefined interface, to be integrated seamlessly into the "conn" API.

This chapter covered the major design goals for a TCP implementation tailored to the IoT and described ways to achieve them. The next chapter covers core aspects of the implementation, based on the previously defined design goals.

7 Implementation

This chapter covers important aspects of the gnrc TCP implementation in detail. In these code listings debug information and minor details were removed to focus on the core functionality. Every code listing contains a comment referencing the original source code file in the RIOT source tree, where the code listing is taken from.

7.1 Generic TCP API

When the gnrc TCP development started, the API for interfacing with the gnrc network stack was not finalized. The current TCP implementation had to supply its own temporary API. Future versions of gnrc TCP have to be integrated into the conn API and its BSD socket API wrapper, the current API is deprecated and should be removed. Code listing 1 contains an overview of the functions currently used to interface with gnrc TCP.

```
/* File: sys/include/net/gnrc/tcp.h */
int8_t gnrc_tcp_tcb_init(gnrc_tcp_tcb_t *tcb);

int8_t gnrc_tcp_tcb_destroy(gnrc_tcp_tcb_t *tcb);

int8_t gnrc_tcp_open(gnrc_tcp_tcb_t *tcb, uint16_t local_port,
                     uint8_t *peer_addr, size_t peer_addr_len,
                     uint16_t peer_port, uint8_t options);

ssize_t gnrc_tcp_send(gnrc_tcp_tcb_t *tcb, uint8_t *buffer,
                      size_t length);

ssize_t gnrc_tcp_recv(gnrc_tcp_tcb_t *tcb, void* buffer,
                      size_t length);

int8_t gnrc_tcp_close(gnrc_tcp_tcb_t *tcb);
```

Code listing 1: Temporary gnrc TCP API

The function *gnrc_tcp_tcb_init()* initializes a TCB and adds it to the linked TCB list. A TCB is removed from the TCB list by calling *gnrc_tcp_tcb_destroy()*. A TCB must be initialized before the TCB is used, otherwise the runtime behavior is unpredictable.

A connection is initiated by calling *gnrc_tcp_open()*. The option parameter determines whether a connection is active or passive. If the option *AI_PASSIVE* is set, the connection is passive, waiting for an incoming connection. In case of a passive connection, a port to listen on must be specified. An active connection needs to supply an IP-address and port number to connect to. In both operation modes the function returns after connection establishment or after occurrence of a user space timeout.

The *gnrc_tcp_send()* function transfers data to the peer. This function expects a connection's TCB, a buffer that contains the data to send and the amount of bytes that should be transmitted. This function blocks until the specified data has been transmitted and acknowledged, or the user space timeout expired.

The *gnrc_tcp_recv()* function, reads the requested amount of data from the receive buffer into the supplied buffer. Like most TCP API functions this function blocks as well, until the requested data has been received and copied or until the user space timeout expires.

The last API function is *gnrc_tcp_close()*. This function triggers the connection termination sequence. It blocks until a connection is closed or until the user space timeout timer occurs. Both outcomes lead to a closed connection.

This API does not conform to the widespread BSD socket API and is temporary. The final version of the gnrc TCP interface should be integrated into the conn API seamlessly and support the BSD socket wrapper to facilitate portability between platforms.

7.2 TCB

Code listing 2 covers the gnrc TCP transmission control block. The TCB contains a connection state, address and port number information, variables for sliding window maintenance, the receive buffer, variables for the retransmission mechanism, a pointer to the next TCB and data structures for synchronization and messaging.

In common TCP implementations the port and address information is stored inside the socket data structure because UDP uses them as well. In gnrc TCP, address and port information used for multiplexing are stored inside the TCB. With the integration of gnrc TCP into the conn API, these variables will be removed from the TCB, they will be stored in the RIOT socket implementation. By containing a receive buffer per connection, the TCB is a large data structure. The receive buffer size can be adjusted by changing the macros in *sys/include/net/gnrc/tcp/config.h*.

The receive buffer size depends on the MSS and the number of MSS-sized payloads the buffer should be able to store.

```c
/* File: sys/include/net/gnrc/tcp/tcb.h */
typedef struct __attribute__((packed)) tcb {
    gnrc_tcp_fsm_state_t state;

    uint8_t*  peer_addr;
    size_t    peer_addr_len;
    uint16_t  peer_port;
    uint16_t  locl_port;
    uint8_t   options;

    uint32_t snd_una;
    uint32_t snd_nxt;
    uint16_t snd_wnd;
    uint32_t snd_wl1;
    uint32_t snd_wl2;
    uint32_t iss;
    uint32_t rcv_nxt;
    uint16_t rcv_wnd;
    uint32_t irs;
    uint16_t mss;

    uint8_t buf[GNRC_TCP_RCV_BUF_SIZE];
    ringbuffer_t rcv_buf;

    gnrc_pktsnip_t* pkt_retransmit;
    xtimer_t timer_timeout;
    msg_t msg_timeout;

    kernel_pid_t owner;
    struct tcb * next;
    mutex_t mtx;
    msg_t msg_queue[TCB_MSG_QUEUE_SIZE];
} gnrc_tcp_tcb_t;
```

Code listing 2: Transmission control block

Buffer size adjustments influence the announced receive window directly. The receive buffer is implemented as a ring buffer using the RIOT ringbuffer library. TCP needs to buffer incoming data until it is consumed by an application. Copying received data into a separate receive buffer is a simple solution, but it increases the memory requirements significantly. To reduce gnrc TCP memory requirements, a future version could store received payload inside the global packet buffer until an application consumes the received data directly from the packet buffer.

The retransmission mechanism of gnrc TCP is lightweight. By allowing only a single packet to be transmitted at a time, only one segment needs to be stored. The TCB stores a pointer to the pre-allocated segment inside the global packet buffer. Additionally a timer and a message structure are necessary to notify the TCP handling thread that a segment needs to be retransmitted. With the introduction of an adaptive retransmission strategy, the TCB has to be extended to store variables for RTT calculation.

Additionally the TCB contains a pointer to another TCB to be a node in a linked list (see 6.4). The remaining variables are used for message passing and synchronization purposes.

7.3 FSM synchronization

The access to the FSM of a connection must be synchronized, different events could be triggered from multiple threads at the same time, therefore the FSM is a critical section. The FSM is locked exclusively by using a mutex per connection, stored inside the TCB. On entry of the FSM function the mutex must be locked, on exit the mutex must be released. The TCP FSM is a complex function, spanning nearly 400 lines of code with multiple exit points. Releasing the mutex before each return statement in the FSM function is error prone and should be avoided. Code listing 3 covers an easy solution to address this problem by implementing a procedural version of the monitor object pattern common in object oriented programming (OOP).

In OOP, access to the methods of an object can be synchronized by implementing the monitor object pattern. A method's core functionality is encapsulated in a private method. Additionally, a public method is added. This method can be called by multiple threads at the same time. On entry of the public method, a mutex must be acquired before the internal private function is called. After return from the internal function, the previously locked mutex is released. This design pattern ensures reliable synchronization, separated from the method's internal function. This pattern can be added to existing code with minimal effort.

By encapsulating the FSM functionality in the static function *_fsm_unprotected()*, this function is protected from external access. Static functions can be called from c-code inside the same file exclusively. The function *_fsm()* is declared externally in the according header file. After including the FSM header file, this function can be called externally. When entering the *_fsm()* function, the mutex of this connection is locked. After locking, the critical section is entered. The critical section covers the entire *_fsm_unprotected()* function.

```
/* File: sys/net/gnrc/transport_layer/gnrc_tcp_fsm.c */
static int32_t _fsm_unprotected(gnrc_tcp_tcb_t *tcb,
                                gnrc_tcp_fsm_event_t event)
{
    /* ... FSM core function resides here ... */
    return 0;
}

int32_t  _fsm(gnrc_tcp_tcb_t *tcb, gnrc_tcp_fsm_event_t event)
{
    int32_t result;

    mutex_lock(&tcb->mtx);
    result = _fsm_unprotected(tcb, event);
    mutex_unlock(&tcb->mtx);

    return result;
}
```

Code listing 3: FSM function protected by procedural monitor object pattern

By leaving the function, the mutex is unlocked automatically. The next pending thread can enter the critical section.

7.4 TCP connection handling

All TCP connections are processed in a single thread. By running a RIOT project with the module *GNRC_TCP* set, the RIOT auto-initialization function starts the connection handling thread on startup. Code listing 4 shows the event loop of the connection handling thread. It uses the core facilities of the gnrc network stack and integrates seamlessly into the message passing based communication between network layers.

After initializing the variables the TCP handling thread registers itself in the network registry “netreg“. Every time a TCP related message is sent, the TCP handling thread receives a message containing a message type and a pointer to a packet inside the packet buffer. If a message with type *GNRC_NETAPI_MSG_TYPE_RCV* is received, the packet was sent from the lower network layer to the TCP layer. The receive function is called with the pointer to the packet in response. If the received message type is *GNRC_NETAPI_MSG_TYPE_SND*, the application layer will have sent a packet to the TCP thread before, to pass the message down the network stack.

```
/* File: sys/net/gnrc/transport_layer/gnrc_tcp_eventloop.c */
void *_event_loop(__attribute__((unused)) void *arg)
{
  gnrc_netreg_entry_t entry;
  entry.demux_ctx = GNRC_NETREG_DEMUX_CTX_ALL;
  entry.pid = _tcp_pid;
  gnrc_netreg_register(GNRC_NETTYPE_TCP, &entry);

  while(1){
    msg_receive(&msg);
    switch (msg.type) {
      case GNRC_NETAPI_MSG_TYPE_RCV:
        _receive((gnrc_pktsnip_t *)msg.content.ptr);
        break;

      case GNRC_NETAPI_MSG_TYPE_SND:
        _send((gnrc_pktsnip_t *)msg.content.ptr);
        break;

      case MSG_TYPE_RETRANSMISSION:
        _fsm((gnrc_tcp_tcb_t *)msg.content.ptr, TIMEOUT_RETRANSMIT,
             NULL, NULL, 0);
        break;

      case MSG_TYPE_TIMEWAIT:
        _fsm((gnrc_tcp_tcb_t *)msg.content.ptr, TIMEOUT_TIMEWAIT,
             NULL, NULL, 0);
        break;
      }
  }
  return NULL;
}
```

Code listing 4: Event loop of the TCP connection handling thread

In addition to the netapi message passing scheme, the event loop processes messages sent on timer expiration. An expired retransmission timer sends a message with *MSG_TYPE_RETRANSMISSION* and a pointer to the connections TCB to the TCP handling thread. On reception of the message, the FSM can be called directly with the affected TCB. The same mechanics apply on reception of a message with type *MSG_TYPE_TIMEWAIT*. This message-based network processing scheme is deployed in every independent layer of the gnrc network stack. By relaying purely on IPC mechanisms, the network layers are independent from each other. New layers can be integrated easily without changing other layers. Code listing 5 covers the *_receive()* function called by the event loop on packet reception.

```c
/* File: sys/net/gnrc/transport_layer/gnrc_tcp_eventloop.c */
static int8_t _receive(gnrc_pktsnip_t *pkt)
{
    /* Get IP and TCP Header */
    LL_SEARCH_SCALAR(pkt, ip6, type, GNRC_NETTYPE_IPV6);
    assert(ip6 != NULL);
    LL_SEARCH_SCALAR(pkt, tcp, type, GNRC_NETTYPE_TCP);
    assert(tcp != NULL);

    /* Extract control bits */
    ctl = byteorder_ntohs(((tcp_hdr_t*) tcp->data)->off_ctl);

    /* Validate Offset */
    if(GET_OFFSET(ctl) < OPTION_OFFSET_BASE){
        gnrc_pktbuf_release(pkt);
        return -ERANGE;
    }

    hdr = (tcp_hdr_t *)tcp->data;

    /* Validate Checksum */
    if(byteorder_ntohs(hdr->checksum)
    != _pkt_calc_csum(tcp, ip6, pkt)
    ){
        gnrc_pktbuf_release(pkt);
        return -EINVAL;
    }

    /* De-multiplex connection */
    syn = ((ctl & MSK_SYN_ACK) == MSK_SYN) ? true : false;
    src = byteorder_ntohs(hdr->src_port);
    dst = byteorder_ntohs(hdr->dst_port);

    /* Iterate over TCB list to find the connections TCB */
    tcb = _head_list_tcb;
    while(tcb){
        /* If SYN is set and a TCB is waiting for a connection */
        if(syn && tcb->locl_port == dst && tcb->state == LISTEN){
            _fsm(tcb, RCVD_PKT, pkt, NULL, 0);
            break;
        }
        /* If port numbers match and SYN is not set */
        if(!syn && tcb->locl_port == dst && tcb->peer_port == src){
            _fsm(tcb, RCVD_PKT, pkt, NULL, 0);
            break;
        }
        tcb = tcb->next;
    }
    /* No fitting TCB has been found. Respond with reset */
    if(tcb == NULL){
        if((ctl & MSK_RST) != MSK_RST){
            _pkt_build_reset_from_pkt(&rst, pkt);
            gnrc_netapi_send(_tcp_pid, rst);
        }
        return -ENOTCONN;
    }
    gnrc_pktbuf_release(pkt);
    return 0;
}
```

Code listing 5: Receive function of the TCP thread

At the beginning, the function searches for the IP and TCP headers inside the received packet. Both are necessary for later checksum calculation and verification. As soon as the TCP header was found, the "offset"-field from the TCP header is extracted and verified. A TCP header with an offset less than 5 is faulty and discarded. After offset verification, the checksum is calculated and compared to the received checksum for transmission error detection. If the calculated checksum matches the received checksum, the received packet will be de-multiplexed, otherwise the packet is discarded.

During de-multiplexing, the TCP handling thread iterates over the TCB list until a TCB associated with the received packet is found. If the received packet contains a SYN-flag and an application is listening on the destination port of the received packet, that TCB can accept the connection. If the SYN flag is not set and the port numbers of the received packet match with the port numbers stored in a TCB, the packet belongs to a connection specified with this TCB. This connections FSM will be called with the *RCVD_PKT* event and a pointer to the received packet. In case of unsuccessful de-multiplexing, a packet with the RST-flag set is sent to the peer in response. Further verification of the sequence and acknowledgement numbers are applied inside the FSM.

7.5 Sequence number processing

Sequence number validation verifies that a received sequence number falls into the receive window. The receive window is defined as interval between *RCV_NXT* and *RCV_NXT + RCV_WND*. A received sequence number is acceptable if the following equation is fulfilled:

$$\mathit{RCV\_NXT} \leq seqno. < \mathit{RCV\_NXT} + \mathit{RCV\_WND}$$

In the natural number space $\mathbb{N}_0$, the verification can be achieved with normal comparison operators. However, sequence numbers are represented as unsigned 32-bit integers forming a subset of $\mathbb{N}_0$. In general the amount of transmitted data in a TCP stream is not limited. The number of transmitted bytes could exceed the limited sequence number space, ranging from 0 to $2^{32} - 1$, therefore arithmetical operations on sequence numbers are calculated modulo 2^{32} implicitly.

As a result of this restricted number space, normal comparison operators can not be used. A calculation of *RCV_NXT + RCV_WND* could overflow, comparisons with $<$ and $>$ can lead to false results. This problem is addressed by introducing overflow tolerant comparison operators. The operators are defined by the following relations:

$$(x < y) <=> ((x - y) < 0)$$

$$(x > y) <=> ((x - y) > 0)$$

Code listing number 6 contains overflow tolerant comparison operators. Every comparison between sequence numbers should use these macros.

```
/* File: sys/net/gnrc/transport_layer/gnrc_tcp_pkt.c */
#define LSS_32_BIT(x, y) (((int32_t) (x)) - ((int32_t) (y)) <  0)
#define LEQ_32_BIT(x, y) (((int32_t) (x)) - ((int32_t) (y)) <= 0)
#define GRT_32_BIT(x, y) (!LEQ_32_BIT(x,y))
#define GEQ_32_BIT(x, y) (!LSS_32_BOT(x,y))
```

Code listing 6: Overflow tolerant comparison operators

7.6 Current implementation status

The gnrc TCP development has progressed to a functional prototype. The prototype can be tested and verified, although the implementation still lacks some functionality necessary for widespread deployment. The missing features can be divided into two categories. The first category is mandatory features for TCP operation. The second category is optional features, improving TCP performance, usability and the integration into gnrc network stack.

The mandatory features cover:

- Zero window probing to detect a reopening of a closed window.
- An adaptive retransmission scheme based on RTO calculations, see[PACS11].
- Testing and verification on multiple hardware platforms.

Optional features:

- Piggybagging payload data onto outgoing acknowledgments.
- Integration into conn API.
- Integration into BSD socket wrapper based on the conn API.
- Integration into the interactive RIOT shell.

The next chapter covers the testing and verification procedure of gnrc TCP. The test setup and different test scenarios are described in detail and the measured results are visualized and explained.

8 Testing

The purpose of a transport protocol is reliable data exchange between applications, independent of the underlying operating system. Therefore the TCP test and verification procedure must verify base functionality and interoperability between nodes running RIOT-OS and nodes running other operating systems.

8.1 Test methodology

For testing and verification purposes a test application was developed. The test application follows the specific sequence of events:

1. A connection is established between a client and a server.
2. The client requests a predefined amount of data from the server.
3. The server reads the request and sends the requested data to the client.
4. After successful data reception, client and server close the connection simultaneously.

The test application client and server were implemented as RIOT applications using gnrc TCP and on Linux. The developed test applications allow three different test scenarios:

1. Scenario: Communication between a RIOT server and a RIOT client.
2. Scenario: Communication between a RIOT server and a Linux client.
3. Scenario: Communication between a Linux server and a RIOT client.

In each scenario, the client application requests 4031 bytes from the server. The gnrc TCP implementation uses a 1220 byte MSS, the RIOT server must split the data stream into multiple segments and the receiving RIOT client is forced to close and reopen its receive window multiple times. The test procedure covers basic TCP operation between two hosts with absence of packet loss. The server component addressed by fc00::1 always listens on port 2000, the client connects to [fc00::1]:2000 from fc00::2 with a randomly generated port. The behavior in case of packet loss is not covered by this test scenario.

8.2 Test environment

The gnrc TCP implementation has been developed and tested under Arch Linux. RIOT features executable and linking format (ELF) as compilation target. ELF can be executed on most Unix-like platforms like Linux or FreeBSD. By running as a normal process in a Linux environment, common c development tools and compiler features can be used for debugging and testing purposes. Furthermore, Linux features virtual tap devices. Tap devices are virtual network interfaces used to connect local processes. A tap device behaves like a normal layer 2 Ethernet device, multiple tap interfaces can be connected to form a virtual network.

The client and server test applications were implemented as RIOT and Linux versions to cover "the three" different test scenarios. Each application is executed as ELF binary running under Linux, the applications are connected by tap devices. The packet sniffer "wireshark"[16], version 2.0.1, is used for capturing the network traffic as well as verification of the transmitted data segments. The tests were executed under Arch Linux, version 4.3.3-3-ARCH. Tap devices do not feature packet loss simulation, this setup is not feasible to simulate a lossy environment and behavior under real world conditions.

8.3 Scenario 1: RIOT as server and RIOT as client

The TCP relevant measurement results for RIOT to RIOT communication are listed in Table 3. The connection is established by performing the 3-Way-Handshake (No. 7 to 9). The client sends SYN-Flag carrying segment (No. 7). The server answers with a SYN+ACK carrying packet (No. 8). The client acknowledges the reception of the server's packet with an ACK (No. 9).

After connection establishment, the client requests 4031 bytes from the server (No. 10). This request covers 20 bytes in size. Packet No. 11 acknowledges the request reception by sending an acknowledgement with an updated window of 1200 byte. The server application consumes the received data and sends an additional acknowledgement, reopening the window to its original size of 1220 bytes (No. 12).

The first segment of the requested data is sent to the client (No. 13). The payload size is 1220 bytes, filling the clients receive window entirely. On segment reception, the client responds with an acknowledgement closing the receive window (No. 14). After consumption of the accepted payload, the client reopens the window (No. 15). This procedure repeats until the requested data has been transmitted (No. 16 - No. 24).

[16] `https://www.wireshark.org/`

No	Source	Destination	Protocol	Length	Info
7	fc00::2	fc00::1	TCP	78	48730 > 2000 [SYN] Seq=0 Win=1220 Len=0 MSS=1220
8	fc00::1	fc00::2	TCP	78	2000 > 48730 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
9	fc00::2	fc00::1	TCP	74	48730 > 2000 [ACK] Seq=1 Ack=1 Win=1220 Len=0
10	fc00::2	fc00::1	TCP	94	48730 > 2000 [ACK] Seq=1 Ack=1 Win=1220 Len=20
11	fc00::1	fc00::2	TCP	74	2000 > 48730 [ACK] Seq=1 Ack=21 Win=1200 Len=0
12	fc00::1	fc00::2	TCP	74	[TCP Window Update] 2000 > 48730 [ACK] Seq=1 Ack=21 Win=1220 Len=0
13	fc00::1	fc00::2	TCP	1294	[TCP Window Full] 2000 > 48730 [ACK] Seq=1 Ack=21 Win=1220 Len=1220
14	fc00::2	fc00::1	TCP	74	[TCP ZeroWindow] 48730 > 2000 [ACK] Seq=21 Ack=1221 Win=0 Len=0
15	fc00::2	fc00::1	TCP	74	[TCP Window Update] 48730 > 2000 [ACK] Seq=21 Ack=1221 Win=1220 Len=0
16	fc00::1	fc00::2	TCP	1294	[TCP Window Full] 2000 > 48730 [ACK] Seq=1221 Ack=21 Win=1220 Len=1220
17	fc00::2	fc00::1	TCP	74	[TCP ZeroWindow] 48730 > 2000 [ACK] Seq=21 Ack=2441 Win=0 Len=0
18	fc00::2	fc00::1	TCP	74	[TCP Window Update] 48730 > 2000 [ACK] Seq=21 Ack=2441 Win=1220 Len=0
19	fc00::1	fc00::2	TCP	1294	[TCP Window Full] 2000 > 48730 [ACK] Seq=2441 Ack=21 Win=1220 Len=1220
20	fc00::2	fc00::1	TCP	74	[TCP ZeroWindow] 48730 > 2000 [ACK] Seq=21 Ack=3661 Win=0 Len=0
21	fc00::2	fc00::1	TCP	74	[TCP Window Update] 48730 > 2000 [ACK] Seq=21 Ack=3661 Win=1220 Len=0
22	fc00::1	fc00::2	TCP	445	2000 > 48730 [ACK] Seq=3661 Ack=21 Win=1220 Len=371
23	fc00::2	fc00::1	TCP	74	48730 > 2000 [ACK] Seq=21 Ack=4032 Win=849 Len=0
24	fc00::2	fc00::1	TCP	74	[TCP Window Update] 48730 > 2000 [ACK] Seq=21 Ack=4032 Win=1220 Len=0
25	fc00::2	fc00::1	TCP	74	48730 > 2000 [FIN, ACK] Seq=21 Ack=4032 Win=1220 Len=0
26	fc00::1	fc00::2	TCP	74	2000 > 48730 [ACK] Seq=4032 Ack=22 Win=1220 Len=0
27	fc00::1	fc00::2	TCP	74	2000 > 48730 [FIN, ACK] Seq=4032 Ack=22 Win=1220 Len=0
28	fc00::2	fc00::1	TCP	74	48730 > 2000 [ACK] Seq=22 Ack=4033 Win=1220 Len=0

Table 3: Network measurement results between a RIOT server and a RIOT client

The connection termination phase stretches from packet no. 25 to no. 28. The client sends a FIN+ACK packet to the server (No. 25), the server acknowledges it (No. 26) and sends its own FIN+ACK packet (No. 27). Finally the server's termination request is acknowledged by the peer (No. 28).

8.4 Scenario 2: RIOT as server and Linux as client

In this scenario the RIOT application acts as server and the Linux application as client. Table 4 contains the measured network traffic.

The connection is initiated by the client sending a SYN packet (No. 4). The Linux client announces a large window of 28800 bytes and a maximum segment size of 1440 bytes. Advanced features like SACK are supported by the full-featured TCP implementation of Linux, a SACK permit option is sent on connection establishment. The RIOT server acknowledges the received SYN with a SYN+ACK packet (No. 5). By not receiving a SACK permit option the client concludes that SACK is not implemented by the peer, SACK will not be used in this connection. The client acknowledges the received SYN+ACK (No. 6), the connection is established.

The client requests data from the server (No. 7), the RIOT server acknowledges the request (No.8) and updates the window after data consumption (No. 9). The transmission of the requested 4031 bytes from the RIOT server to the Linux client is straight forward. The server

No	Source	Destination	Protocol	Length	Info
4	fc00::2	fc00::1	TCP	94	43316 > 2000 [SYN] Seq=0 Win=28800 Len=0 MSS=1440 SACK_PERM=1
5	fc00::1	fc00::2	TCP	78	2000 > 43316 [SYN, ACK] Seq=0 Ack=1 Win=1220 Len=0 MSS=1220
6	fc00::2	fc00::1	TCP	74	43316 > 2000 [ACK] Seq=1 Ack=1 Win=28800 Len=0
7	fc00::2	fc00::1	TCP	94	43316 > 2000 [PSH, ACK] Seq=1 Ack=1 Win=28800 Len=20
8	fc00::1	fc00::2	TCP	74	2000 > 43316 [ACK] Seq=1 Ack=21 Win=1200 Len=0
9	fc00::1	fc00::2	TCP	74	[TCP Window Update] 2000 > 43316 [ACK] Seq=1 Ack=21 Win=1220 Len=0
10	fc00::1	fc00::2	TCP	1294	2000 > 43316 [ACK] Seq=1 Ack=21 Win=1220 Len=1220
11	fc00::2	fc00::1	TCP	74	43316 > 2000 [ACK] Seq=21 Ack=1221 Win=30500 Len=0
12	fc00::1	fc00::2	TCP	1294	2000 > 43316 [ACK] Seq=1221 Ack=21 Win=1220 Len=1220
13	fc00::2	fc00::1	TCP	74	43316 > 2000 [ACK] Seq=21 Ack=2441 Win=34160 Len=0
14	fc00::1	fc00::2	TCP	1294	2000 > 43316 [ACK] Seq=2441 Ack=21 Win=1220 Len=1220
15	fc00::2	fc00::1	TCP	74	43316 > 2000 [ACK] Seq=21 Ack=3661 Win=36600 Len=0
16	fc00::1	fc00::2	TCP	445	2000 > 43316 [ACK] Seq=3661 Ack=21 Win=1220 Len=371
17	fc00::2	fc00::1	TCP	74	43316 > 2000 [ACK] Seq=21 Ack=4032 Win=39040 Len=0
18	fc00::2	fc00::1	TCP	74	43316 > 2000 [FIN, ACK] Seq=21 Ack=4032 Win=39040 Len=0
19	fc00::1	fc00::2	TCP	74	2000 > 43316 [ACK] Seq=4032 Ack=22 Win=1220 Len=0
20	fc00::1	fc00::2	TCP	74	2000 > 43316 [FIN, ACK] Seq=4032 Ack=22 Win=1220 Len=0
21	fc00::2	fc00::1	TCP	74	43316 > 2000 [ACK] Seq=22 Ack=4033 Win=39040 Len=0

Table 4: Network measurement results between a RIOT server and a Linux client

sends a segment, waits for the clients acknowledgement and sends the next segment until the payload has been transmitted (No. 10 to No. 17). The client's window is large enough to store the requested payload without closing the window. In theory the server could send multiple and larger segments at one time, because the client announced a MSS of 1440 bytes on connection establishment. Currently the gnrc TCP implementation sends only one segment with 1220 bytes payload at most, to prevent overflows in static packet buffer of the gnrc network stack.

The connection termination is similar to the previous test scenario (No. 18 to No. 21).

8.5 Scenario 3: Linux as server and RIOT as client

In the third test scenario, the server is Linux based and the RIOT application is the client. Table 5 contains the captured network traffic. The connection establishment phase (No. 5 to No. 7) and transmission of the client's request (No. 8 to No. 9) are similar to the other test cases.

The Linux server transmits the requested payload split into multiple 610 bytes segments (No. 10 to 21). Each received segment is acknowledged by the RIOT based client. After receiving two 610 segments, the client's receive buffer is filled (No. 11), closing the receive window (No. 13) until the received data is consumed. After reopening the window (No. 14), the server continues the transmission.

The connection termination is similar to the other test cases with one exception. The FIN packet sent from the Linux server (No. 25) carries payload. This causes the client to acknowledge the data and the FIN-flag in one acknowledgment (No. 26). After consumption of the received

No	Source	Destination	Protocol	Length	Info
5	fc00::2	fc00::1	TCP	78	5354 > 2000 [SYN] Seq=0 Win=1220 Len=0 MSS=1220
6	fc00::1	fc00::2	TCP	78	2000 > 5354 [SYN, ACK] Seq=0 Ack=1 Win=28800 Len=0 MSS=1440
7	fc00::2	fc00::1	TCP	74	5354 > 2000 [ACK] Seq=1 Ack=1 Win=1220 Len=0
8	fc00::2	fc00::1	TCP	94	5354 > 2000 [ACK] Seq=1 Ack=1 Win=1220 Len=20
9	fc00::1	fc00::2	TCP	74	2000 > 5354 [ACK] Seq=1 Ack=21 Win=28800 Len=0
10	fc00::1	fc00::2	TCP	684	2000 > 5354 [ACK] Seq=1 Ack=21 Win=28800 Len=610
11	fc00::1	fc00::2	TCP	684	[TCP Window Full] 2000 > 5354 [PSH, ACK] Seq=611 Ack=21 Win=28800 Len=610
12	fc00::2	fc00::1	TCP	74	5354 > 2000 [ACK] Seq=21 Ack=611 Win=610 Len=0
13	fc00::2	fc00::1	TCP	74	[TCP ZeroWindow] 5354 > 2000 [ACK] Seq=21 Ack=1221 Win=0 Len=0
14	fc00::2	fc00::1	TCP	74	[TCP Window Update] 5354 > 2000 [ACK] Seq=21 Ack=1221 Win=1220 Len=0
15	fc00::1	fc00::2	TCP	684	2000 > 5354 [ACK] Seq=1221 Ack=21 Win=28800 Len=610
16	fc00::1	fc00::2	TCP	684	[TCP Window Full] 2000 > 5354 [PSH, ACK] Seq=1831 Ack=21 Win=28800 Len=610
17	fc00::2	fc00::1	TCP	74	5354 > 2000 [ACK] Seq=21 Ack=1831 Win=610 Len=0
18	fc00::2	fc00::1	TCP	74	[TCP ZeroWindow] 5354 > 2000 [ACK] Seq=21 Ack=2441 Win=0 Len=0
19	fc00::2	fc00::1	TCP	74	[TCP Window Update] 5354 > 2000 [ACK] Seq=21 Ack=2441 Win=1220 Len=0
20	fc00::1	fc00::2	TCP	684	2000 > 5354 [ACK] Seq=2441 Ack=21 Win=28800 Len=610
21	fc00::1	fc00::2	TCP	684	[TCP Window Full] 2000 > 5354 [PSH, ACK] Seq=3051 Ack=21 Win=28800 Len=610
22	fc00::2	fc00::1	TCP	74	5354 > 2000 [ACK] Seq=21 Ack=3051 Win=610 Len=0
23	fc00::2	fc00::1	TCP	74	[TCP ZeroWindow] 5354 > 2000 [ACK] Seq=21 Ack=3661 Win=0 Len=0
24	fc00::2	fc00::1	TCP	74	[TCP Window Update] 5354 > 2000 [ACK] Seq=21 Ack=3661 Win=1220 Len=0
25	fc00::1	fc00::2	TCP	445	2000 > 5354 [FIN, PSH, ACK] Seq=3661 Ack=21 Win=28800 Len=371
26	fc00::2	fc00::1	TCP	74	5354 > 2000 [ACK] Seq=21 Ack=4033 Win=849 Len=0
27	fc00::2	fc00::1	TCP	74	[TCP Window Update] 5354 > 2000 [ACK] Seq=21 Ack=4033 Win=1220 Len=0
28	fc00::2	fc00::1	TCP	74	5354 > 2000 [FIN, ACK] Seq=21 Ack=4033 Win=1220 Len=0
29	fc00::1	fc00::2	TCP	74	2000 > 5354 [ACK] Seq=4033 Ack=22 Win=28800 Len=0

Table 5: Network measurement results between a RIOT server and a Linux client

data, the client sends an unnecessary window update (No. 27). The window update has no effect, because the server has nothing left to send after sending a FIN-flag. This is not an error, it is just unnecessary.

8.6 Memory usage of gnrc TCP

This section covers the memory usage of gnrc TCP. The server component of the test application was compiled for the ARM Cortex-m0+ CPU architecture and analyzed with cosy[17] a tool for analyzing memory usage and distribution in ELF binaries. Table 6 contains the measured memory requirements. The ELF binary was compiled and analyzed with and without compiler optimization (optimization levels O1, O2, O3 and Os). The default optimization level of the RIOT build process is Os (optimization for space usage).

[17] https://github.com/haukepetersen/cosy

	ROM (.text)	RAM (.data + .bss)
Cortex-m0+	13964 bytes	1030 bytes
Cortex-m0+ optimized(-O1)	8112 bytes	1030 bytes
Cortex-m0+ optimized(-O2)	7968 bytes	1030 bytes
Cortex-m0+ optimized(-O3)	8064 bytes	1030 bytes
Cortex-m0+ optimized(-Os)	6930 bytes	1030 bytes

Table 6: Memory usage of gnrc TCP

The numbers in table 6 cover the basic memory requirements of the gnrc TCP module. Additionally, each connection requires a TCB measuring 1394 bytes in size. A space usage optimized application with a single TCP connection, requires additional ROM of 6930 bytes and RAM of 2424 bytes (1030 bytes + 1394 bytes) for TCP operation. The TCB size is constant regardless of the selected optimization level. A detailed analysis of memory usage at runtime and measuring data transfer rates is beyond the scope of this thesis.

9 Conclusion and Outlook

The increasing interconnection between embedded devices, desktop computers, mobile devices and network services requires re-implementations of widespread network protocols tailored to different device classes. The task of implementing protocols that have been extended and optimized for fast data transfer rather than for a low memory footprint can be difficult. In general, protocols and their extensions must be analyzed with respect to the field of application. TCP implementations used in the IoT should supply a reasonable feature selection to minimize system requirements while providing enough functionality to communicate with full-featured TCP implementations (see chapter 5).

In this thesis, the TCP extension SACK and the intervened algorithms slow start (4.2.1), congestion avoidance (4.2.2), fast retransmit (4.2.3) and fast recovery (4.2.4) have been studied with focus on their applicability in an IoT environment (4.2.5). SACK seems to be unsuitable for the IoT (4.1). It increases memory requirements and message size in general. Slow start, congestion avoidance and fast recovery are designed for TCP implementations that send multiple segments in a burst. This is an uncommon behavior in the IoT. The deployment of fast retransmit can lead to a faster retransmission of lost segments and increase throughput in general. As part of this thesis, TCP has been implemented as contribution to the gnrc network stack of RIOT-OS.

The TCP implementation for gnrc is a functional prototype with some missing features. The first measurements of RIOT-to-RIOT and RIOT-to-Linux communication are promising, but there is still room for improvement in terms of memory efficiency and data transmission rates.

9.1 Field of application

By offering reliable data transport, TCP is the basis for many applications. TCP in RIOT might be the basis for a network based interactive shell. Currently, RIOT offers a shell accessible via a serial interface. By deploying a network-based shell, nodes could be controlled from anywhere removing the need to have direct access to the device. Another application could be an update mechanism, a feature missing in most operating systems deployed in the IoT. This is especially important because network access if often the only accessible interface to deployed embedded devices. Additionally, TCP-dependent protocols can be implemented easily. For instance, a simple HTTP server delivering a static website could be implemented with ease. Or nodes could access information from the web extracting information needed for device operation.

9.2 Future work

The gnrc TCP implementation is still an early prototype, missing features like zero window probing and an adaptive retransmission scheme must be added before gnrc TCP can be deployed. Additionally, TCP must be integrated into the conn API. The testing and verification procedures need to be extended to scenarios with lossy networks and real hardware. In the future, gnrc TCP should be analyzed and compared to other TCP implementations like lwIP or µIP in terms of performance, memory requirements and feature completeness.

In the long run, efforts to reduce memory consumption should be taken. By removing the separate receive buffer of a connection, memory requirements could be reduced significantly. Extensions like fast retransmit or a transmission policy allowing concurrent network traversal of multiple segments could improve transmission rates.

To summarized this thesis, the gnrc TCP implementation has the potential to become one of the cornerstones for new developments and applications in the RIOT-OS ecosystem.

References

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

[BBP88] R.T. Braden, D.A. Borman, and C. Partridge. Computing the Internet checksum. RFC 1071, IETF, September 1988.

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

[BHG$^+$13] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C. Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *Proc. of the 32nd IEEE INFOCOM. Poster*, Piscataway, NJ, USA, 2013. IEEE Press.

[Bra89] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, IETF, October 1989.

[CHS14] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications. In *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!*, pages 15–28, New York, NY, USA, Oct. 2014. ACM.

[Cra98] M. Crawford. Transmission of IPv6 Packets over Ethernet Networks. RFC 2464, IETF, December 1998.

[Dee89] S.E. Deering. Host extensions for IP multicasting. RFC 1112, IETF, August 1989.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, December 1998.

[Dun01] Adam Dunkels. Design and implementation of the lwip tcp/ip stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[HT11] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282, IETF, September 2011.

[Jac88] V. Jacobson. Congestion Avoidance and Control. *Computer Communication Review*, 18(4):314–329, Aug 1988.

[JBB92] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, IETF, May 1992.

[MDM96] J. McCann, S. Deering, and J. Mogul. Path MTU Discovery for IP version 6. RFC 1981, IETF, August 1996.

[MKHC07] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, IETF, September 2007.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, IETF, October 1996.

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

[NNSS07] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861, IETF, September 2007.

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

[PLW$^+$15] Hauke Petersen, Martine Lenders, Matthias Wählisch, Oliver Hahm, and Emmanuel Baccelli. Old Wine in New Skins? Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices. In *1st Int. Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys15)*, Florence, Italy, May 2015. ACM.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793, IETF, September 1981.

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

[SB09] Zack Shelby and Carsten Bormann. *6LoWPAN The wireless embedded Internet*. WILEY, 2009.

[SCNB12] Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann. Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 6775, IETF, November 2012.

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

[Sim94] W. Simpson. The Point-to-Point Protocol (PPP). RFC 1661, IETF, July 1994.

[Soc11] IEEE Computer Society. Part 15.4:Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANS): Amendment to Add Alternate Phy. Amendment of IEEE Std 802.15.4, 2011.

[SSZV07] Feng Shu, T. Sakurai, M. Zukerman, and H.L. Vu. Packet loss analysis of the IEEE 802.15.4 MAC without acknowledgements. *Communications Letters, IEEE*, 11(1), Jan 2007.

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

[SWF10] T. Schmidt, M. Waehlisch, and G. Fairhurst. Multicast Mobility in Mobile IP Version 6 (MIPv6): Problem Statement and Brief Survey. RFC 5757, IETF, February 2010.

[TNJ07] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862, IETF, September 2007.

Abbreviations

6LoWPAN	IPv6 over low-power wireless area networks
AES	Advanced encryption Standard
API	Application Programming Interface
BSD	Berkeley Software Distribution
CAF	C++ Actor Framework
CPU	Central Processing Unit
CSMA/CA	Carrier sense multiple access with collision avoidance
ELF	Executable and Linking Format
FSM	Finite State Machine
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-process communication
ISN	Explicit Congestion Notification
ISN	Initial Sequence Number
LGPLv2.1	GNU Lesser General Public License, Version 2.1

LoWPAN	Low-power wireless area network
MSL	Maximum Segment Lifetime
MSS	Maximum Segment Size
MTU	Maximum transmission unit
OOP	Object Oriented Programming
OSI	Open Systems Interconnection
PPP	Point-to-Point Protocol
RAM	Random-Access Memory
RFC	Requests For Comment
RIOT-OS	Real-time Internet of Things Operating System
ROM	Read Only Memory
RTT	Round Trip Time
SACK	Selective Acknowledgments
SICS	Swedish Institute of Computer Science
SSH	Secure Shell
TCB	Transmission Control Block
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 1. April 2016 Simon Brummer