EasyChair Preprint

№ 2434

Towards Securing the Internet of Things with QUIC

Lars Eggert

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

January 21, 2020

Towards Securing the Internet of Things with QUIC

Lars Eggert
NetApp
lars@netapp.com

Abstract—**This paper is the first to evaluate the feasibility of deploying QUIC, a new UDP-based transport protocol currently undergoing IETF standardization, directly on resource-constrained IoT devices. It quantifies the storage, compute, memory and energy requirements of the Quant QUIC stack on two different IoT platforms, and finds that a minimal standards-compliant QUIC client currently requires approximately 58 to 63 KB of flash and can retrieve 5 KB of data in 4.2 to 5.1 s over 0-RTT or 1-RTT connections, using less than 16 KB of heap memory (plus packet buffers), less than 4 KB of stack memory and less than 0.9 J of energy per transaction.**

I. Introduction

QUIC is a secure new transport protocol for the general Internet. Originally designed by Google [1] and deployed in the Chromium browser and on Google's backend servers since 2013 [2], the original "gQUIC" variant was the starting point for the standard version of QUIC that the Internet Engineering Task Force (IETF) is likely to publish in 2020. QUIC—which is *no longer* an acronym [3]—runs over the User Datagram Protocol (UDP) to traverse network address translators (NATs) and firewalls, borrows congestion control algorithms and related concepts [4] from the Transmission Control Procotol (TCP), and uses the latest Transport Layer Security protocol version 1.3 (TLS13) [5] to provide strong, low-overhead security [6]. It multiplexes many concurrent application-level streams into a single end-to-end connection, like the Stream Control Transmission Protocol (SCTP) [7], and will add advanced features in upcoming versions, such as partial reliability (also inspired by SCTP) or multipath operation similar to Multipath TCP (MPTCP) [8].

The initial focus of QUIC is to securely and efficiently carry Hypertext Transfer Protocol (HTTP) traffic; HTTP-over-QUIC has been dubbed "HTTP/3" [9] to differentiate it from HTTP/2 [10], which operates over TCP and Transport Layer Security (TLS) version 1.2 but offers nearly-identical application-level semantics. However, QUIC is a general-purpose transport protocol and likely to replace TCP and TLS for many other applications.

Because of their enormous importance to the smooth operation of the Internet, QUIC and TLS13 have been and continue to be the subject of intense scrutiny. For TLS13, the IETF changed their standardization process [11] to incorporate results from parallel security analyses to harden the standard. One of several such examples corrected an interaction attack based on an automated analysis of the protocol [12]. The ongoing standardization of QUIC and its resulting implementations are similarly informed by concurrent academic work [13, 14].

It would be of great benefit to the Internet of Things (IoT) if the research and engineering effort spent on securing the web protocol suite could be leveraged, because effective and lightweight security mechanisms for IoT remain a key challenge. It is not necessary to implement the complete web public key infrastructure (Web PKI); a subset of the concepts and mechanisms might already bring substantial benefits. Compared to plaintext application protocols on top of UDP, QUIC offers strong security, Internet-ready congestion control, rich applications semantics (streams, soon partial reliability and multipath operation) and has many rapidly maturing implementations [15].

Severely constrained microcontrollers with only a few KB of random-access memory (RAM) and a few tens of KB of persistent storage will likely continue to require carefully tailored, minimal mechanisms. It is unlikely that these low-end "Class 0" devices [16] will be able to adopt even a select few of the key components of the Web PKI. However, due to Moore's law, many more capable devices with tens of KB of RAM and a few hundreds of KB of persistent storage have become available at interesting price points. This paper investigates whether these somewhat more powerful devices have sufficient capabilities to exchange data using QUIC. Specifically, it quantifies the storage, compute, memory and energy requirements of the Quant QUIC stack on two different IoT platforms.

Section II describes the hardware and system software used in this paper in more detail. Section III discusses the Quant QUIC stack and its underlying Warpcore packet I/O framework ported to the two IoT platforms. Section IV presents measurements of the storage, compute, memory and energy requirements of running QUIC on IoT devices. Section V discusses related work, and Section VI concludes this paper.

II. System Hardware and Software

The two IoT platforms chosen for this paper are the Particle Argon [17] and an ESP32-DevKitC V4 development kit [18], referred to in the rest of the paper as "Argon" and "ESP32". Table I compares key attributes of the two platforms.

The Argon runs Particle's Device OS system software [19], which is built on top of an underlying FreeRTOS [20]; release 1.4.3 is used for the experiments in this paper. Device OS is quite bulky, leaving only about 60 KB RAM (out of 256 KB) and 125 KB flash memory (out of 1 MB[1]) available for an

Network and Distributed Systems Security (NDSS) Symposium 2020
23–26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

[1]The Argon includes an additional 4 MB of flash that is accessible over an SPI bus, but that can only be used to store data, not code (this memory is used to emulate the EEPROM).

Table I. IoT PLATFORMS USED FOR THE MEASUREMENTS.

Platform	Particle Argon	ESP32-DevKitC V4
SoC	Nordic Semiconductor nRF52840	ESP32-D0WDQ6
CPU	ARM Cortex-M4F	Tensilica Xtensa LX6
Instruction set	32-bit	32-bit
Clock speed	64 MHz	240 MHz
FPU	IEEE 754 single-precision	IEEE 754 single-precision
Hardware crypto	ARM TrustZone CryptoCell-310	AES, SHA, RSA, and ECC
RAM	256 KB	520 KB
Flash	1 MB (+ 4 MB SPI)	4 MB
Other memory	4 KB EEPROM (emulated)	96 B e-Fuse
WLAN	IEEE 802.11 b/g/n	IEEE 802.11 b/g/n
OS	Device OS 1.4.3	RIOT-OS 2019.10
Toolchain	arm-none-eabi-gcc 5.3.1	xtensa-esp32-elf-gcc 5.2.0

application on the Argon. This is due to maintaining background connectivity to Particle's device cloud, which allows remote maintenance and access to devices, including over-the-air firmware updates. It is unfortunately impossible to disable device cloud connectivity without major changes to Device OS.

The ESP32 normally runs Espressif's "IoT Development Framework" (ESP-IDF), which is also FreeRTOS-based. To investigate whether different underlying IoT OSs affect the feasibility of running QUIC, we chose instead to run the ESP32 under RIOT [21], a completely different OS that shares no code with FreeRTOS. RIOT is also an attractive platform, because it supports much a more diverse set of hardware; currently around 170 different boards of about a dozen different architectures. In the future, we would like to evaluate QUIC on other boards.

These two devices were chosen, because they represent examples of the top end of current resource-constrained devices; at least the class that is not powerful enough to run general-purpose OSs such as Linux. They also both support WLAN connectivity, which eases experimentation.

III. Quant and Warpcore

Quant [22] is a BSD-licensed C implementation of the emerging IETF standard for QUIC. Quant uses Warpcore [23] as its network stack and focuses on high-performance datacenter networking. However, Quant and Warpcore are reasonably lightweight at ca. 10 300 and 3700 lines of code[2], respectively.

A. Warpcore

Warpcore is a minimal BSD-licensed C implementation of an userspace UDP/IP/Ethernet stack for the `netmap` [24] packet I/O framework. It prioritizes performance over features, and over full standards compliance. It supports zero-copy send and receive operations, implementing a run-to-completion model that uses neither threads, timers nor signals. Warpcore also has a backend implementation for the POSIX Socket application programming interface (API), allowing applications linking against the Warpcore API to work without `netmap`.

Device OS on the Argon includes the lightweight IP stack (LWIP) [25], which has enough POSIX compliance to support the Warpcore Socket backend without major modifications[3].

Although RIOT claims to support LWIP as an add-on module, it does not function at present. RIOT's native generic network stack (GNRC) [26] is not POSIX compliant, IPv6-only and lacks core features, such as the ability to "poll", i.e., block execution while waiting for network input. Nevertheless, a bare-bones Warpcore backend was implemented, to evaluate QUIC on RIOT at least to some degree.

Warpcore exports an API that has a socket-like concept for an endpoint of a network session, but I/O operations occur on chains of pre-allocated packet payload buffers, allowing for zero-copy I/O. Neither LWIP nor GNRC support zero-copy I/O, which is unfortunate for IoT OSs, given its benefits in terms of computational and memory overheads.

B. Quant

Quant implements the QUIC transport layer but does *not* implement an HTTP/3 binding (which is a future work, see Section V). It implements an event-based architecture around a tickless implementation [27] of hierarchical timing wheels [28]. Quant uses `khash`, a generic hash table based on double hashing adopted from `klib` [29] to track connections by identifier and for other purposes; `khash` was patched to avoid double-precision math and to allow static allocation. Quant also implements discrete interval encoding trees (DIETs) [30] to concisely represent integer ranges, such as used for QUIC's acknowledgment (ACK) blocks. The DIET implementation is based on splay trees [31], which are also used as a general ordered data structure for some other purposes.

Stemming from Quant's focus on high-speed datacenter networking, these data structures are fast, but *not* particularly optimized for constrained nodes. For example, they are not particular frugal in their memory usage and rely on dynamic memory allocation. Nevertheless, their footprint is sufficiently small for IoT use. One topic for future work (see Section V) is to replace these data structures with more suitable ones.

Besides Warpcore, the most important external dependency of Quant is `picotls` [32], an MIT-licensed C implementation of TLS13 supporting multiple engines that supply the necessary cryptographic primitives. Its "minicrypto" engine is particularly suited for embedded use, relying on `micro-ecc` [33] for a reasonably fast `secp256r1` implementation [34], and on `cifra` [35] for most other cryptographic operations (`X25519`, `AES`, etc.) Although both the Argon and the ESP32 have hardware implementations of most if not all required cryptographic operations needed by TLS13 (see Table I), neither OS makes them available to applications, and `picotls` also does not have an engine that would exploit their presence on either platform.

One critical limitation of the `picotls` "minicrypto" engine is that it is currently missing support for verifying the peer signature during the TLS handshake. This is a temporary implementation limitation and not a limitation on the use of TLS on constrained nodes; appropriate profiles exist [36].

IV. Measurement Results

This section presents a several initial measurement results to quantify the storage, compute, memory and energy footprints of the Quant QUIC stack on the Argon and ESP32.

[2]Plus ca. 3000 for `netmap` support (not compiled in on IoT platforms).

[3]As part of this research, a few patches were contributed to Device OS to expose some required LWIP calls, such as `poll`.

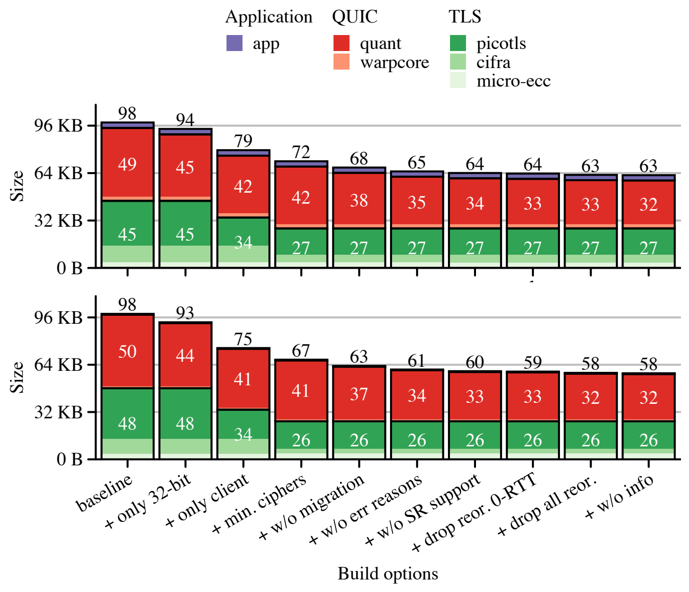

Figure 1. Argon (top) and ESP32 (bottom) code sizes for different build options, cumulative from left to right.

A. *Code Size*

We first investigate the static footprint of Quant and its dependencies, i.e., we try to answer the question of how much flash memory is required to hold their code and static data.

Figure 1 shows two bar diagrams—the top one for the Argon platform, the bottom for the ESP32—that quantify the build sizes of Quant under a combination of different build options, which are cumulatively applied left-to-right. In addition to reporting the overall binary size (on top of the respective bars), the diagrams also show the sizes of Quant's main components, i.e., a minimal application[4] performing a download of a 5 KB object over WLAN, Quant and Warpcore, which make up the main QUIC component, and `picotls`, `cifra` and `micro-ecc`, which make up the TLS component. Not shown in Figure 1 are the OS sizes for the two platforms, which weigh in at around 604 KB for the Argon[5] and 267 KB for the ESP32.

Although both platforms claim support for link time optimization (LTO), their use of outdated toolchains from 2015/16 (see Table I) with various LTO bugs can result in link errors or non-executable binaries. The reported numbers have therefore been obtained with LTO *disabled*. When the platforms switch to newer toolchains with functioning LTO, minor additional reductions in footprint are possible. For example, on the ESP32 (where the build at least succeeds with LTO enabled), the resulting binaries shrink by about 2 %.

The leftmost "baseline" bars show that a minimal application using Quant requires 98 KB of flash on both on the Argon and the ESP32, without any particular build options in addition to the toolchain defaults (both already set `-Os` to optimize for size, `-DNDEBUG` to disable assertions, etc.) Also, Quant by default performs only single-precision floating point operations, which are handled by the hardware on both platforms and therefore do not require helper functions that increase the binary size further (as double-precision math would.)

The next "32-bit only" bars quantify the sizes of the build when the costliest 64-bit operations have been eliminated, resulting in around 4 to 5 KB of reductions. A large part is contributed by the avoidance of 64-bit divisions, all occurrences of which divide by constant values and can therefore approximated by much cheaper multiplications with "magic" numbers, followed by right-shifts [37]. Other savings accrue from using 32-bit values (and math) for several internal QUIC variables, such as packet numbers, window sizes and round-trip time (RTT)-related measures (which are kept in µs).

By default, a Quant application can act as a QUIC server and client at the same time (to enable proxying, etc.) Dropping support for QUIC server mode ("only client" bars) yields an additional 3 KB saving at the QUIC level over the "32-bit only" builds, but a much more significant 11 to 14 KB reduction at the TLS level. TLS13 is a complex and sufficiently asymmetric protocol, and the removal of all server-side processing from the binary is a big saver. The more modest savings at the QUIC layer are mostly due to using zero-length connection identifiers (IDs), which allow removal of a several data structures servers need for ID-based lookups.

TLS13 requires implementing the `AES_128_GCM_SHA256` cipher suite and recommends the implementation of both `AES_256_GCM_SHA384` and `CHACHA20_POLY1305_SHA256`. Likewise, it requires `secp256r1` as a key exchange, and recommends implementation of `X25519` [5]. For IoT use, especially if software updates are possible, it is likely enough to only implement the minimum subset of those, i.e., the `AES_128_GCM_SHA256` cipher suite and the `secp256r1` key exchange. The "min. ciphers" bars show that this leads to a further large 7 to 8 KB code size reduction compared to the "client-only" measurement, specifically, to the `cifra` component. Note that *if* Device OS or RIOT were able to expose the crypto facilities present in the hardware, `cifra` and possibly `micro-ecc` could be dropped completely, yielding reductions of 5 KB and 4 KB, respectively.

Together, the build options above reduce the footprint of Quant by about 25 to 30 %. It is possible to make some further gains by giving up support for some features of QUIC, but these gains are comparatively minor:

- QUIC features "connection migration", which allows a client to switch an established QUIC connection to a path through a different interface. This is mostly useful for multi-homed clients and long-lived connections, neither of which is likely in an IoT scenario. The "w/o migration" bars quantify the resulting build sizes when support for migration is not compiled into Quant; a further reduction of about 3 KB.
- QUIC's `CONNECTION_CLOSE` frames contain optional error reasons, which are human-readable strings, and Quant by default uses them for quite verbose messages. Dropping the transmission of error reasons reduce the sizes of the builds by 1 to 2 KB, as shown by the "w/o err reasons" bars, because these static strings are no longer part of the binary.
- A "stateless reset" can be sent by one endpoint that has already dropped state for a QUIC connection to signal this

[4]The application bar is not visible in the ESP32 diagram, because the code size is only 483 B. The Argon application has more features, such as reacting to button presses and signaling status via LEDs, and weighs in at 3 KB.

[5]RIOT is a bare-bones OS that only includes the minimal functionality required by the application. Device OS has many more features, such as connecting the device to Particle's cloud, over-the-air firmware updates, etc.

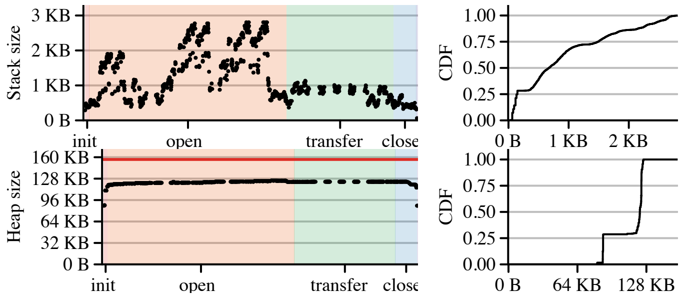

Figure 2. Argon stack and heap sizes during a 5 KB download.

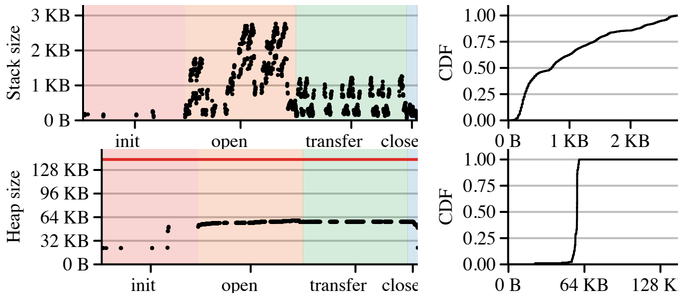

Figure 3. ESP32 stack and heap sizes during a 5 KB download.

in an authenticated way to its peer. This is an optimization, and not supporting stateless resets reduces the footprint of Quant by another 1 KB ("w/o SR support" bars).

- Packet reordering is an occasional occurrence in many networks. By default, Quant caches `STREAM` and `CRYPTO` frame data that arrives out-of-order, to save the peer from retransmitting it. However, the data structures to support this caching have a cost and dropping reordered zero-RTT data or all reordered data can avoid that cost. Doing so reduces the footprint by 1 KB when dropping 0-RTT only ("drop reor. 0-RTT" bars), and another 1 KB when dropping all reordered data ("drop all reor." bars).
- Quant by default collects connection statistics, similar to the `TCP_INFO` structure in Linux [38]. Disabling this functionality reduces the footprint by up to another 1 KB ("w/o info" bars).

Overall, a minimal Quant client requires 63 KB flash on the Argon and 58 KB on the ESP32, a reduction of 36 % and 40 % over the initially measured baseline size, respectively.

B. *Stack and Heap Requirements*

This section quantifies the stack and heap memory requirements of running Quant on the Argon and the ESP32. To that end, each device is requesting a 5 KB object from a server on the same WLAN. The binaries running on the IoT devices are instrumented (via `-finstrument-functions`) so that stack size, heap size and the call stack depth is logged over a serial connection whenever a function is entered or exited in the application, Quant, Warpcore and `picotls` modules.

Due to the high overheads of logging this data over a slow 112.5 Kbit/s link, it is impractical to also instrument the `cifra` and `mirco-ecc` cryptographic libraries, which have many small functions that greatly increase the logging overhead. It is also not possible to store this data on-device, due to its volume. The data reported in this section are therefore *not* the maximum stack and heap sizes, which are several hundred bytes higher. Nevertheless, the data gives a reasonable understanding of the progression of stack and heap memory usage and should give a reasonably accurate picture of what the stack usage will be when the cryptographic operations are offloaded to hardware.

Figure 2 and Figure 3 show the progression of the stack sizes (top diagrams) and heap sizes (bottom diagrams) for the Argon and the ESP32, respectively[6]. Also shown are cumulative distribution functions (CDFs) of the raw data, to the right of the respective time series plots. Note that time units are *not* shown on purpose; the overheads of logging are so high[7] that their absolute values are meaningless. Each plot is subdivided into several phases: the "init" phase (red background) covers Quant and Warpcore initialization, the "open" phase (orange background) covers the establishment of a QUIC connection to the server, the "transfer" phase (green background) covers the sending of the request and the reception of the 5 KB response, and the "close" phase (blue background) covers closing down the QUIC connection and releasing the resources allocated in the "init" phase. Furthermore, the red horizontal line in the heap plots shows the maximum available heap memory of the given platform. Gaps in the time series occur either when waiting for data from the peer, or when the non-instrumented cryptographic modules or the OS execute.

The stack usage in Figures 2 and 3 is quite similar, apart from the much longer "init" phase[8] in Figure 3. In both figures, stack usage during "open" almost reaches 3 KB, whereas it remains around 1 KB during the "transfer" phase. The public key cryptography involved in the TLS handshake obviously requires more stack than the symmetric key cryptography used during the data transfer. Unfortunately, what matters is *maximum* stack usage during the execution, and a value of 3 KB is large for resource-constrained devices. Much of this usage occurs in `picotls`, which uses several statically allocated buffers of several hundred bytes (see Section V).

The heap usage in Figures 2 and 3 stays rather flat throughout the execution, which is desirable behavior on resource-constrained nodes. A main reason for this is that Quant preallocates packet buffers during the "init" phase; this can be observed in Figure 3, where the heap usage jumps by about 23 KB, which corresponds to the allocation of 15 maximum transmission unit (MTU)-sized packet buffers[9]. Another 8 KB are allocated for packet meta-data, an MTU-sized scratch space for various en- and decoding operations, and other global data. During the transfer, the heap size drops by another 4.5 KB due to dynamic data structure allocation (ACK blocks, etc.)

Although the Argon reports more available heap (around 156 KB) compared to the ESP (around 142 KB), Device OS uses more heap memory for its execution compared to RIOT,

[6]In the diagrams on the left, a random 20 % of data points are plotted, to reduce overplotting.

[7]Even exporting the reported data took tens of seconds.

[8]For the ESP32, "init" includes associating to the WLAN, whereas the Argon performs this procedure during boot (before instrumentation begins), hence the much longer "init" phase.

[9]The same is somewhat visible on the very left of Figure 2

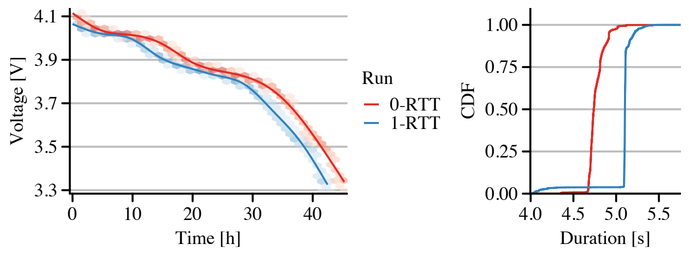

Figure 4. Argon battery voltage (left) and transaction durations (right) for repeated 1-RTT and 0-RTT connections.

leaving less for the application. On the Argon, heap usage is around 128 KB, while it is around 64 KB on the ESP32.

C. Energy and Performance

This section quantifies the energy consumption and overall performance of executing QUIC transactions on resource-constrained nodes. Due to the lack of a suitable battery for the ESP32, this experiment was only performed on the Argon. It is powered by an—initially fully charged—2000 mAh 3.7 V lithium-ion polymer (LiPo) battery holding about 26 640 J of energy. The Argon then continuously requests 5 KB objects from a server located on the same WLAN, thereby slowly draining the battery. Two separate runs are performed: one where all connections perform a full 1-RTT handshake, and another where all but the first connection instance use an accelerated 0-RTT handshake. The two diagrams in Figure 4 show the results of these experiments.

The left diagram in Figure 4 plots battery voltage over time for both the 1-RTT and 0-RTT runs. From a starting voltage of approximately 4.1 V, the Argon drains its battery over the course of about 42 to 45 h until its charge is insufficient to power the device. Each battery charge is enough for 29 338 1-RTT and 31 844 0-RTT connections; each transaction takes around 0.90 J and 0.83 J, respectively. Due to the coarseness of the Argon's battery level reporting, more experiments are needed to verify whether this difference is due to a somewhat higher initial charge at the beginning of the 0-RTT run or if 0-RTT connections do in fact consume measurably less energy.

The right diagram in in Figure 4 is a CDF of the transaction durations for the 1-RTT and 0-RTT run. The median 1-RTT connection took 5.10 s compared to the median 0-RTT connection at 4.74 s, an 8 % improvement.

V. Future and Related Work

To the best of our knowledge, this paper is the first to investigate the feasibility of deploying QUIC directly on resource constrained IoT devices. Another recent paper [39] investigated using QUIC to transport Message Queuing Telemetry Transport (MQTT) [40], by running a subset of the Google's original Chromium "gQUIC" code on Raspberry Pi Model 3B devices. However, those are so much more powerful (quad-core ARMv8 Cortex A53 at 1.2 GHz, 1 GB RAM, many GB of storage) than either of the two platforms investigated in this paper that is questionable whether they even qualify as resource-constrained.

QUIC itself [1] has been the topic of several papers, focusing on its deployment and performance on the Internet [41, 42], especially for media traffic [43, 44, 45] and some of its core components, such as its security [46] and multipath [47, 48] frameworks.

In the broader IoT space, a several communication protocols have been proposed; a recent paper [49] compares several of the key ones, including the Constrained Application Protocol (CoAP) [50], the Advanced Message Queuing Protocol (AMQP) [51] and MQTT [40].

The experimental results in Section IV should be expanded to evaluate the performance of QUIC more fully. Specifically, data upload cases should be measured in addition to the existing measurements for downloads, and measurements should try to quantify the overheads and performance of various QUIC mechanisms, such as streams, frames, etc. Experiments should vary the sizes of transferred objects, and the performance of QUIC should be compared against plain TCP, TLS over TCP as well as other IoT protocols such as CoAP or MQTT. Finally, once the RIOT stack gains support for asynchronous I/O, a performance comparison between the Argon and the ESP32—or RIOT on other boards—may prove interesting.

Several other development-oriented items are future work:

- Quant currently implements the QUIC transport protocol [3, 4, 6] but not its HTTP/3 binding [9]. A future version of Quant should optionally include support for HTTP/3, either by depending on an existing implementation [52] or by creating its own. An investigation like the one in Section IV should quantify the resulting overheads.
- One of the largest consumers of runtime stack is `picotls`. It should be straightforward to at least end its use of large stack-allocated buffers, which may allow QUIC to support more constrained hardware more easily.
- The hash and splay data structures in Quant are not particularly suitable for embedded use. First, their implementation as generic C macros causes code duplication when they hold multiple different date types. Second, they optimize for managing large numbers of elements efficiently, which is unnecessary on constrained nodes, which will usually only have one or at most a few active connections, and those running at low speeds. More suitable data structures should reduce the binary code size by several more KB.
- Making use of cryptographic operations provided by the hardware, rather than relying on `cifra` and `micro-ecc`, has the potential to reduce the required flash size, as well as improve performance and reduce energy consumption.
- Since the preliminary energy measurements in Section IV-C do not show significant energy benefits of using 0-RTT, a future study should investigate if the TLS footprint could be further reduced by dropping 0-RTT support.
- The measurements in the paper were performed over WLAN. IPv6 over Bluetooth Low Energy (BLE) [53] or IEEE 802.15.4 [54] may have different performance/energy consumption behaviors, and hence may allow QUIC to run on more constrained hardware. As a data point, the WLAN driver on the ESP32 weighs in at 115 KB—this is almost half of RIOT's total size (267 KB) on that platform. Link layers designed for IoT use, such as BLE or IEEE 802.15.4 should have smaller footprints.

With the improvements above, it may become possible to consider experimenting with QUIC on 16-bit or even 8-bit microcontrollers. Doing so will however entail other challenges, such as lack of any floating-point support, etc.

VI. Conclusion

This paper presents a feasibility analysis of deploying the upcoming IETF standard for QUIC directly on resource-constrained 32-bit IoT edge devices. It finds that it is possible to deploy a QUIC client in a few tens of KB of flash and RAM, with a reasonable energy consumption and performance. With some further effort, it may become possible to reduce the required footprint to deploy QUIC on 16-bit microcontrollers.

Acknowledgment

Thanks to Gunar Schorcht and Martine Lenders for help with RIOT and GNRC on the ESP32, Andrey Tolstoy for merging patches for improved LWIP support into Particle Device OS, and Kazuho Oku for help with `picotls`. Martin Thomson and Lucas Pardue made other helpful suggestions.

References

[1] A. Langley, A. Riddoch, A. Wilk, *et al.*, "The QUIC transport protocol: Design and Internet-scale deployment," Proc. *ACM SIGCOMM*, Aug. 2017.

[2] I. Swett, *QUIC deployment experience @Google*, Jul. 2016. https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf.

[3] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-24, Nov. 2019, Work in Progress.

[4] J. Iyengar and I. Swett, "QUIC loss detection and congestion control," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-recovery-24, Nov. 2019, Work in Progress.

[5] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," RFC Editor, RFC 8446, Aug. 2018.

[6] M. Thomson and S. Turner, "Using TLS to secure QUIC," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-tls-24, Nov. 2019, Work in Progress.

[7] P. Natarajan, J. R. Iyengar, P. D. Amer, and R. Stewart, "SCTP: An innovative transport layer protocol for the Web," Proc. *WWW*, May 2006.

[8] C. Raiciu, C. Paasch, S. Barré, *et al.*, "How hard can it be? Designing and implementing a deployable multipath TCP," Proc. *USENIX NSDI*, Apr. 2012.

[9] M. Bishop, "Hypertext transfer protocol version 3 (HTTP/3)," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-24, Nov. 2019, Work in Progress.

[10] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2 (HTTP/2)," RFC Editor, RFC 7540, May 2015.

[11] K. Paterson and T. van der Merwe, "Reactive and proactive standardisation of TLS," Proc. *SSR*, Dec. 2016.

[12] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," Proc. *IEEE SSP*, May 2016.

[13] K. L. McMillan and L. D. Zuck, "Formal specification and testing of QUIC," Proc. *ACM SIGCOMM*, Aug. 2019.

[14] F. Rath, D. Schemmel, and K. Wehrle, "Interoperability-guided testing of QUIC implementations using symbolic execution," Proc. *ACM EPIQ*, Dec. 2018.

[15] QUIC implementations, https://github.com/quicwg/base-drafts/wiki/Implementations.

[16] C. Bormann, M. Ersue, and A. Keränen, "Terminology for constrained-node networks," RFC Editor, RFC 7228, May 2014.

[17] Particle, *Argon datasheet (v001)*, Oct. 2018. https://docs.particle.io/assets/pdfs/datasheets/argon-datasheet.pdf.

[18] Espressif Systems, *ESP32 series datasheet (version 3.2)*, Oct. 2019. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[19] Particle. Device OS (firmware) for Particle devices, https://github.com/particle-iot/device-os.

[20] FreeRTOS, https://www.freertos.org/.

[21] E. Baccelli, C. Gündoğan, O. Hahm, *et al.*, "RIOT: An open source operating system for low-end embedded devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, Dec. 2018.

[22] Quant, https://github.com/NTAP/quant.

[23] Warpcore, https://github.com/NTAP/warpcore.

[24] L. Rizzo, "netmap: A novel framework for fast packet I/O," Proc. *USENIX ATC*, Jun. 2012.

[25] A. Dunkels, "Design and implementation of the LWIP TCP/IP stack," Swedish Institute of Computer Science (SICS), Tech. Rep., Feb. 2001.

[26] M. Lenders, "Analysis and comparison of embedded network stacks – design and evaluation of the GNRC network stack," Master's thesis, Freie Universität Berlin, Apr. 2016.

[27] W. Ahern. timeout.c: Tickless hierarchical timing wheel, http://25thandclement.com/~william/projects/timeout.c.html.

[28] G. Varghese and T. Lauck, "Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility," Proc. *ACM SOSP*, Nov. 1987.

[29] Attractive Chaos. klib, https://attractivechaos.github.io/klib/.

[30] M. Erwig, "Diets for fat sets," *Journal of Functional Programming*, vol. 8, 1998.

[31] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, Jul. 1985.

[32] H2O. picotls, https://github.com/h2o/picotls.

[33] K. MacKay. micro-ecc, https://github.com/kmackay/micro-ecc.

[34] T. Silde, "Comparative study of ECC libraries for embedded devices," Norwegian University of Science and Technology, Tech. Rep., Mar. 2019.

[35] J. Birr-Pixton. cifra, https://github.com/ctz/cifra.

[36] H. Tschofenig and T. Fossati, "Transport layer security (TLS) / datagram transport layer security DTLS) profiles for the Internet of Things," RFC Editor, RFC 7925, Jul. 2016.

[37] H. S. Warren, *Hacker's Delight (2nd Edition)*. Oct. 2012.

[38] R. Pfeiffer, "Measuring TCP congestion windows," *Linux Gazette*, Mar. 2012.

[39] P. Kumar and B. Dezfouli, "Implementation and analysis of QUIC for MQTT," *Computer Networks*, vol. 150, Feb. 2019.

[40] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "MQTT version 5.0," OASIS, OASIS Standard, Mar. 2019.

[41] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, "Taking a long look at QUIC: An approach for rigorous evaluation of rapidly evolving transport protocols," Proc. *ACM IMC*, Nov. 2017.

[42] J. Rüth, K. Wolsing, K. Wehrle, and O. Hohlfeld, "Perceiving QUIC: Do users notice or even care?" Proc. *ACM CoNEXT*, Dec. 2019.

[43] D. Bhat, A. Rizk, and M. Zink, "Not so QUIC: A performance study of DASH over QUIC," Proc. *ACM NOSSDAV*, Jun. 2017.

[44] C. Perkins and J. Ott, "Real-time audio-visual media transport over QUIC," Proc. *ACM EPIQ*, Dec. 2018.

[45] M. Palmer, T. Krüger, B. Chandrasekaran, and A. Feldmann, "The QUIC fix for optimal video streaming," Proc. *ACM EPIQ*, Dec. 2018.

[46] M. Hall-Andersen, D. Wong, N. Sullivan, and A. Chator, "nQUIC: Noise-based QUIC packet protection," Proc. *ACM EPIQ*, Dec. 2018.

[47] A. Rabitsch, P. Hurtig, and A. Brunstrom, "A stream-aware multipath QUIC scheduler for heterogeneous paths," Proc. *ACM EPIQ*, Dec. 2018.

[48] Q. De Coninck and O. Bonaventure, "Multipath QUIC: Design and evaluation," Proc. *ACM CoNEXT*, Dec. 2017.

[49] J. Dizdarević, F. Carpio, A. Jukan, and X. Masip-Bruin, "A survey of communication protocols for Internet of Things and related challenges of fog and cloud computing integration," *ACM Comput. Surv.*, vol. 51, Jan. 2019.

[50] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (CoAP)," RFC Editor, RFC 7252, Jun. 2014.

[51] R. Godfrey, D. Ingham, and R. Schloming, "OASIS advanced message queuing protocol (AMQP) version 1.0," OASIS, OASIS Standard, Oct. 2012.

[52] T. Tsujikawa. nghttp3, https://github.com/ngtcp2/nghttp3.

[53] J. Nieminen, T. Savolainen, M. Isomäki, *et al.*, "IPv6 over BLUETOOTH(R) low energy," RFC Editor, RFC 7668, Oct. 2015.

[54] G. Montenegro, N. Kushalnagar, J. W. Hui, and D. E. Culler, "Transmission of IPv6 packets over IEEE 802.15.4 networks," RFC Editor, RFC 4944, Sep. 2007.