QUIC for Constrained Nodes

A Criteria-Based Assessment of Contemporary QUIC Implementations for IoT Devices

Maximilian Brodesser

HAW Hamburg, 20099 Hamburg
maximilian.brodesser@haw-hamburg.de

Abstract. Reusing QUIC on constrained IoT devices would remove the need for a second security stack alongside the DTLS- and OSCORE-based one that CoAP deployments rely on today. The feasibility of such reuse is governed less by the specification than by the individual implementation a device has to run, and more than twenty QUIC stacks are under active development. This paper inventories 22 of them, classifies them by implementation language, cryptographic coupling, I/O model and protocol scope, and screens them against eight criteria derived from the requirements of Class 1 and Class 2 constrained nodes in the sense of RFC 7228. Three stacks pass the screen: *picoquic*, *ngtcp2* and *quicly*, none of them characterised on a microcontroller. The only such characterisation covers *quant*, which the screen rejects as dormant. Recalculating that study against the RFC 7228 budgets yields a minimum of 34 KB of RAM for a reduced client, three times the Class 1 budget. The evidence is graded as low-certainty, and a research agenda is derived that identifies ngtcp2 with wolfSSL on RIOT as the port target. That port will itself require hardware above Class 2, since the operating system and the transport together exceed the Class 2 code budget.

Keywords: QUIC · Internet of Things · Constrained nodes · Transport protocols · Embedded systems

1 Introduction

QUIC [13,31,12] integrates connection establishment with the TLS 1.3 handshake [24], encrypts nearly all of its own control information, multiplexes streams without head-of-line blocking, and tolerates changes of the client's network path. HTTP/3 [3] was defined on top of it.

The Internet of Things has settled on a different set of protocols. Constrained nodes typically speak CoAP [30] over UDP and secure it either with DTLS 1.3 [25] or with object security at the application layer [29,28]. Maintaining this second, largely disjoint stack requires an additional body of code to be audited and a further set of cryptographic decisions to be kept current. Eggert therefore raised the question whether the effort already invested in securing the web transport can be reused instead of duplicated [5].

Whether such reuse is possible is usually discussed as a property of the protocol. It is more accurately a property of the individual implementation, because QUIC leaves a large part of its observable behaviour to the implementer. Stacks conforming to the same drafts acknowledge anywhere between every packet and every tenth, choose initial congestion windows differing by more than a factor of three, and size application data frames from under 100 kB to more than 1 MB [19]. Later throughput studies attribute most of the observed variation to the implementation rather than to the protocol [14,17]. On a device with a few tens of kilobytes of RAM, differences of this magnitude decide whether the software can be deployed at all.

The guiding question is accordingly which QUIC implementations could plausibly be operated on a Class 1 or Class 2 constrained node, and what the published evidence supports. Section 2 restates the problem in terms of the obligations RFC 9000 imposes on an endpoint. Section 3 inventories 22 actively developed implementations, and Sect. 4 defines eight screening criteria and applies them. Section 5 derives a research agenda from the gaps this exposes. The work assesses published evidence and project documentation against those criteria rather than reporting measurements of its own. All quantitative statements originate with other authors and are reported with their conditions of measurement.

2 Problem Statement: QUIC on Constrained Nodes

2.1 Resource Budgets

The terminology of RFC 7228 [4] is used throughout. Class 0 devices ($\ll 10$ KiB RAM, $\ll 100$ KiB flash) cannot communicate securely over the Internet without a proxy. Class 1 devices (~ 10 KiB, ~ 100 KiB) are "quite constrained in code space and processing capabilities" but can participate as IP peers using CoAP, and Class 2 devices (~ 50 KiB, ~ 250 KiB) are "fundamentally capable of supporting most of the same protocol stacks" as ordinary hosts. Every component on the device draws on these budgets, the operating system and the network stack included. The budgets are absolute figures rather than proportions, which is easily overlooked when measurements taken on a single-board computer are presented as results about IoT devices. Of the two, the data budget is the one that binds: code size can be traded against features at compile time, whereas per-connection state cannot.

2.2 Requirements Imposed on an Endpoint

Minimum datagram size. RFC 9000 §14.1 requires a datagram carrying an Initial packet to be at least 1200 bytes long, and §8.1 prohibits an endpoint from sending more than three times the number of bytes it has received from an address that has not yet been validated [13]. Together the two rules establish a lower bound that no implementation may undercut: a client has to emit, and to buffer, at least one datagram of 1200 bytes.

Integrated TLS 1.3 handshake. QUIC provides no unauthenticated mode. Every connection performs an ephemeral key exchange and, in the usual configuration, verifies an X.509 certificate chain [31]. Certificate parsing and signature verification are the largest consumers of stack space and CPU time on a microcontroller, and, being incurred once per connection, they are difficult to amortise on a device that connects only to transmit a single measurement.

Per-connection state and header protection. Loss recovery [12] requires every packet in flight to be retained with its metadata until it is acknowledged or declared lost. Tolerating reordering requires receive buffers, flow control a credit window per stream and per connection, and migration sets of connection IDs and stateless-reset tokens. Each is dimensioned in multiples of the path MTU, and none is optional in a conformant endpoint. QUIC additionally protects every packet header cryptographically on top of the AEAD protection of the payload [31]. This costs one further block-cipher invocation per packet: an AES-ECB operation over a 16-byte sample of the ciphertext, or a single ChaCha20 block. Set against the order of 75 block operations that AEAD performs on a 1200-byte datagram, the addition is marginal, but its relative weight grows as packets shrink, and it requires a second key to be derived and held in RAM.

2.3 Constrained Links

The lower bound of 1200 bytes is difficult to reconcile with constrained link layers. IEEE 802.15.4 frames carry at most 127 bytes, and 6LoWPAN bridges the gap to the 1280-byte IPv6 minimum MTU by header compression and fragmentation [21]. A single QUIC Initial datagram therefore occupies roughly a dozen link-layer fragments, all of which have to arrive before reassembly succeeds, so the probability of a successful handshake falls sharply as the per-frame loss rate rises. A complete EDHOC exchange, by comparison, consists of three messages totalling about 101 bytes, and a DTLS 1.3 handshake with raw public keys about 894 bytes [20]. QUIC's mandatory first datagram alone exceeds both.

2.4 Related Work and the State of the Evidence

Only two pieces of work run QUIC on a microcontroller at all, and neither uses hardware as small as Class 2. Eggert [5] ported the *quant* stack to a Particle Argon (Cortex-M4F, 64 MHz, 256 KB RAM) and to an ESP32 (520 KB RAM) running RIOT [2], boards carrying five and ten times the Class 2 data budget, respectively, and reported flash footprint, stack and heap usage, and energy per transaction. It remains the only peer-reviewed characterisation of its kind and has not been replicated. A vendor proof of concept later ran ngtcp2 with wolfSSL on an ESP32-C3 [7], but disabled certificate verification and reported no comparative measurements.

The remaining five studies use general-purpose platforms. MQTT over QUIC has been reported to reduce handshake packet count, processor utilisation and memory consumption against MQTT over TCP/TLS [18] and to lower delay

Table 1. The 21 endpoint implementations among the 24 entries listed as active by the IETF QUIC Working Group [23], plus quant. Properties from project documentation, August 2026. "n.s." = not stated, "picotls-mc" = picotls with minicrypto, "tlshd" = handshake delegated to a userspace daemon, "Ap-2" = Apache-2.0, † = distributed only within a product.

Impl.	Lang.	TLS/licence	Impl.	Lang.	TLS/licence
aioquic	Python	own/BSD-3	Neqo	Rust	NSS/Ap-2, MIT
AppleQuic†	C, ObjC, Swift	n.s./propr.	ngtcp2	C	8 backends/MIT
Chromium	C, C++	BoringSSL/BSD-3	nginx	C	OpenSSL/BSD-2
f5†	C	n.s./propr.	picoquic	C	picotls/MIT
haproxy	C	OpenSSL/GPL-2+	quant	C11	picotls-mc/BSD
Haskell quic	Haskell	Haskell tls/BSD-3	quiche	Rust	BoringSSL/BSD-2
kwik	Java	own/LGPL	quicly	C	picotls/MIT
linuxquic	C (kern.)	tlshd/GPL-2	Quinn	Rust	rustls/Ap-2, MIT
lsquic	C	BoringSSL/MIT	quic-go	Go	Go stdlib/MIT
MsQuic	C	platform/MIT	s2n-quic	Rust	s2n-tls/Ap-2
mvfst	C++	fizz/MIT	XQUIC	C	2 backends/Ap-2

and jitter at comparable energy [9,15], the latter on a Raspberry Pi 3B, whose 1 GB of RAM is four orders of magnitude above the Class 2 data budget. Jung et al. [16] examined CoAP over QUIC. Saif and Matrawy [27] compared HTTP/3 with MQTT over QUIC under emulated NB-IoT conditions in virtual machines, and found that HTTP/3 saved one round trip but used roughly ten times the heap, with 37.5% of CPU time in X.509 processing. That last finding transfers to constrained nodes. The resource figures, from a four-core virtual machine, do not.

3 The Landscape of QUIC Implementations

3.1 Inventory

The register of active implementations and tools maintained by the IETF QUIC Working Group serves as the sampling frame [23]. Being maintained by the standards body, it limits selection bias in favour of well-marketed stacks. Since entries are added by the projects themselves, unregistered stacks do not appear and dormant ones are not removed. Of its 24 entries, three are tooling rather than endpoint implementations and are excluded here: qlog/qvis, the Interop Runner itself, and Wireshark. One stack the register omits is added: *quant*, the subject of the only published microcontroller study. This yields the 22 entries of Table 1.

3.2 Structural Dimensions

Implementation language. Thirteen of the stacks are written in C or C++, four in Rust, four in languages with a managed runtime (Go, Python, Java, Haskell),

and one, AppleQuic, in a mixture of C, Objective-C and Swift. The third group can be set aside on quantitative grounds alone, since the runtime of a garbage-collected language, before any QUIC code, occupies megabytes, at least an order of magnitude beyond the entire Class 2 data budget.

Cryptographic coupling. Most stacks are bound to a single TLS library: BoringSSL for quiche and lsquic, NSS for Neqo, fizz for mvfst, and the Go standard library for quic-go. Six are less tightly coupled: eight backends for ngtcp2, among them wolfSSL [22,32], a pluggable cryptographic provider for Quinn, either BoringSSL or BabaSSL for XQUIC [1], and picotls for picoquic, quant and quicly. Since flash footprint and handshake CPU time on a microcontroller are usually dominated by the TLS library and not by the transport, backend substitutability is one of the more consequential properties in the table.

I/O model and protocol scope. Implementations that own their event loop and socket handling are difficult to integrate into an embedded RTOS. MsQuic creates its own worker threads and drives the UDP datapath from them, and mvfst depends on folly. Implementations whose protocol core is a pure state machine are easier to port, whatever the language: quiche and `quinn-proto` leave I/O to the application entirely, so it is their runtime rather than their architecture that rules them out here. quant ships its own userspace UDP/IP stack, *warpcore*, through which the port to RIOT's GNRC was realised [5]. ngtcp2 delegates HTTP/3 to a companion library, an arrangement that suits an endpoint carrying CBOR payloads over QUIC streams.

Unintended divergence. Marx et al. [19] additionally found that only 3 of 14 stacks implemented any form of path MTU discovery and that three violated the 3× amplification limit, one by a factor of 36. Both defects sit in mechanisms a constrained port would have to modify.

4 Assessment

4.1 Criteria

Eight criteria are derived from Sect. 2, each stated so that it could in principle be falsified by measurement. They fall into two groups. A *required* criterion (R) is decidable from project documentation for every stack in Table 1 and cannot be engineered around, so failing one removes a stack from consideration. A *graded* criterion (G) decides nothing: G1 and G2 need measurements no project publishes, and G3 concerns code size, which Sect. 4.4 shows to be governed by the platform rather than the stack. Graded criteria structure the comparison among the survivors, which is why a dash appears in a G column of Table 2 but never in an R column.

R1 Toolchain fit: the implementation targets Cortex-M or RISC-V in a bare-metal or RTOS build, with neither a hosted language runtime nor a general-purpose operating system beneath it. **R2 Cryptographic substitutability:**

the TLS backend is replaceable by a library suitable for constrained nodes and able to delegate to a hardware accelerator. A stack already built on such a library satisfies the criterion without being substitutable. **R3 I/O decoupling:** the protocol core operates without threads, POSIX sockets or a specific event loop. **R4 Maintenance and conformance:** the project is under active development, with conformance evidenced by interoperability testing. **R5 Licence:** use in unpublished firmware is permitted. **G1 Static footprint:** transport and cryptography, excluding the operating system, fit the class code budget alongside the application. **G2 Dynamic footprint:** peak RAM, comprising pre-allocated buffers, per-connection state and worst-case stack depth, fits the class data budget, preferably as bounded static allocation. **G3 Feature reducibility:** server role, HTTP/3, migration and surplus cipher suites are removable at compile time.

4.2 Screening

R1 excludes nine stacks. quic-go, aioquic, kwik and Haskell quic require a managed runtime, and linuxquic is not a portable library but a Linux kernel subsystem that exposes an `IPPROTO_QUIC` socket and delegates the handshake to a userspace daemon. None of quiche, Neqo, s2n-quic and Quinn documents support for `no_std` targets, so each of the four needs the Rust standard library and with it a hosted platform. The exclusion follows from the resource budgets and implies no judgement of quality.

R2 removes lsquic, whose build requires BoringSSL and documents no alternative, and XQUIC, whose pluggable cryptography offers only BoringSSL and BabaSSL, neither of which fits a Class 2 node. It also counts against quiche and Neqo, excluded above, which need BoringSSL built for the target and NSS.

R3 removes MsQuic and mvfst. MsQuic creates its own threads by default, one per processor and affinitised to a NUMA node, and drives the UDP datapath from them. The variant in which the application supplies the threads still expects IOCP, epoll or kqueue. mvfst depends on folly and fizz and is built around a multi-threaded UDP socket server with a thread-local architecture. s2n-quic, already excluded under R1, performs its own I/O as well and requires a Linux kernel of version 5.0 or later.

R4 removes quant. It is the one stack in the inventory with published microcontroller measurements, on which Sect. 4.4 rests, but its repository has seen no development for two years [6], the register of active implementations does not list it, and its author reports that the internal data structures were selected for datacentre workloads and are “not particularly suitable for embedded use” [5]. It is therefore treated here as evidence rather than as a candidate. R5 removes none of the survivors, though it would independently rule out the GPL-licensed entries of Table 1 for unpublished firmware.

The remaining five fall outside the comparison rather than failing a criterion: Chromium quiche, nginx and haproxy are maintained as parts of larger products rather than as portable libraries, and AppleQuic and f5 are not separately obtainable.

Table 2. Assessment of the remaining candidates. • = met on published or documentary evidence, ○ = plausible but unevidenced, − = not met. No candidate has a measured peak-RAM figure, so the G2 marks record only whether bounded static allocation is architecturally available.

Stack	R1	R2	R3	R4	R5	G1	G2	G3	Confidence
picoquic	•	○	○	•	•	○	○	○	very low
ngtcp2	•	•	•	•	•	○	•	○	very low
quicly	•	○	•	○	•	○	−	−	very low

Three candidates remain: **picoquic**, **ngtcp2** and **quicly** (Table 2). Screen and evidence base are therefore disjoint: none of the three is the stack that has been measured.

4.3 The Remaining Candidates

picoquic is a minimalist C implementation intended to provide feedback during standardisation and to support applications other than HTTP [11]. It is the most thoroughly tested candidate, and a throughput gain of 44.8% between 2023 and 2025 [17] documents continuous maintenance, though directed at a workload other than the one considered here. The default build requires OpenSSL, which no Class 2 node can accommodate. The remaining picotls path is bounded by the minicrypto engine, which, by the project's own documentation, can sign a handshake with the certificate key "but cannot verify a signature sent by the peer" [10]. This is the gap Eggert already encountered in 2020, and it is unresolved. A binding to mbedTLS, which would close it, appears in picoquic's build documentation as an open issue rather than as a shipped backend [11].

ngtcp2 is the strongest candidate on structural grounds. Its core library has no external dependencies, it is I/O-agnostic, HTTP/3 is provided separately, and eight TLS backends are supported, among them wolfSSL [22], which targets embedded platforms, supports hardware accelerators [32] and ships as an upstream RIOT package [26]. It is also the only candidate exposing a pluggable allocator: `ngtcp2_mem` carries the four allocation functions with a `user_data` pointer, and the header names the purpose, "to achieve per-connection memory pool" [22]. State that scales with load is bounded through the transport parameters for stream count, connection-ID count and flow-control windows. The ESP32-C3 demonstration [7] is the only public microcontroller port since Eggert's work, but it reports 1.6 MB including the entire ESP-IDF and Wi-Fi stack and has not been reproduced.

quicly was written for use inside the H2O server, which says nothing about its portability: the register lists it as a library, and its public API is strictly datagram-oriented. Packets enter through `quicly_receive` and leave through `quicly_send`, which writes into a caller-supplied buffer, so the library never touches a socket and does not own the send buffer. It builds on picotls and exposes a crypto-engine hook intended for hardware offload. Its weaknesses lie

elsewhere: no allocator hook, no compile-time feature switches and no client-only build, and no public interop server, which leaves its conformance evidence thinner than that of picoquic and ngtcp2. Being built on picotls, it inherits the verification gap described above.

4.4 Recalculation Against the RFC 7228 Budgets

Eggert's measurements are the only ones that can be related to the RFC 7228 budgets. Because both boards are themselves above Class 2, what follows is a projection onto smaller hardware than was tested rather than a report of a device that fits. The artefact is quant [6]: about 10 300 lines plus 3 700 for its own UDP/IP stack *warpcore*, on picotls with the minicrypto engine and without HTTP/3. Eggert reports kilobytes where RFC 7228 uses kibibytes, a 2.4% difference below the precision of either figure.

In terms of *code size*, a baseline build of 98 KB was reduced to 58 KB on the ESP32 and 63 KB on the Argon by 32-bit-only arithmetic, removal of the server role and restriction to AES_128_GCM_SHA256 with secp256r1, which lies within the Class 1 code budget of ~ 100 KiB. The same paper reports, however, 267 KB of operating-system code for RIOT outside the application binary, so the complete system exceeds even the Class 2 code budget by a wide margin. The limiting component is the platform rather than the transport.

The *data* budget yields a different picture. quant pre-allocates about 23 KB of packet buffers, 15 of MTU size, each bounded from below by the 1200-byte rule of Sect. 2. A further 8 KB go to packet metadata and scratch space, and peak stack usage during the handshake reaches about 3 KB [5]. The resulting total of at least 34 KB is a lower bound, since the cryptographic libraries were excluded from instrumentation. Against the class budgets that is about 3.4 times the Class 1 budget and two thirds of the Class 2 budget, before operating system, network stack and application.

The energy figures require a more cautious interpretation than they are usually given. The reported 0.90 J per 1-RTT and 0.83 J per 0-RTT transaction were obtained by discharging a 2000 mAh battery to exhaustion and dividing, so the 8% difference rests on two runs with coarse fuel-gauge resolution and is compatible with an advantage of 0-RTT operation without establishing one. Of greater interest is the median connection time of 5.10 s for a 5 KB object over WLAN. The source does not decompose it, so link setup, the userspace network stack, retransmission timers and asymmetric cryptography on a 64 MHz core without accelerator access remain unseparated. Which of them dominates is an open question rather than a finding.

4.5 Threats to Validity

The evidence base is thin and unreplicated. The claim that QUIC can run on a microcontroller rests on one workshop paper by one author covering two boards, and on a stack the screen rejects. That is low-certainty evidence under any

established grading scheme, and no candidate in Table 2 reaches even that grade, none having been measured at all.

The measured artefact was not deployable. It could not verify peer signatures and was built against a pre-RFC draft, so the reported figures are lower bounds for a system not yet implemented.

Comparators are absent. No study on constrained hardware compares QUIC with DTLS 1.3 or OSCORE on the same device, so feasibility cannot be converted into preferability, although the designer's decision is comparative.

Publication bias. Successful ports are reported whereas abandoned attempts are not, and interoperability testing establishes the presence of features rather than correct behaviour under constraint.

The conclusion the evidence supports is accordingly narrower than the formulation common in the literature. A reduced QUIC client has been fitted onto a microcontroller once, on hardware above Class 2 and in a configuration that was not security-complete. Projected onto a Class 2 budget it would claim at least two thirds of the available RAM. The same projection puts Class 1 and Class 0 out of reach, though no candidate has been measured to confirm it, and no published evidence establishes that QUIC is preferable to DTLS 1.3 or OSCORE on a constrained node.

5 Conclusion and Outlook

Of 22 actively developed QUIC implementations, nine are excluded by their runtime, five by the way they are distributed, four by cryptographic coupling or I/O architecture and one by dormancy. That leaves picoquic, ngtcp2 and quicly, none of them characterised on a microcontroller. The only such study covers quant, excluded here for dormancy, on hardware above Class 2. Projected onto the RFC 7228 budgets it rules out Class 1 devices and leaves roughly a third of the Class 2 data budget for everything else. No measurements exist for ngtcp2 with wolfSSL, the port target adopted below.

5.1 Outlook I: A Measurement Campaign

Questions. **RQ1:** What are the flash, static RAM, peak stack and peak heap footprints of a security-complete RFC 9000 client built from ngtcp2 with wolfSSL, and how do picoquic and quicly on picotls compare once the missing peer-signature verification is supplied? **RQ2:** How do these footprints and the energy per transaction compare with CoAP over DTLS 1.3 and CoAP with OSCORE on the same device? **RQ3:** How does the energy of a QUIC session decompose into handshake, transfer and idle maintenance, and at which reporting interval does 0-RTT resumption become cheaper than a fresh handshake? **RQ4:** How does handshake success degrade as the link MTU approaches the 6LoWPAN case?

Design. A factorial design is proposed over stack (ngtcp2, picoquic and quicly, against CoAP over DTLS 1.3 and CoAP with OSCORE), link technology (802.11, 802.15.4/6LoWPAN, BLE, NB-IoT), payload size and session mode. It should run under RIOT on both an nRF52840 and an ESP32-C3, so that accelerator availability becomes an observable factor rather than a confounder. Peak stack should be read by watermark painting rather than function instrumentation, whose overhead forced Eggert to exclude the cryptographic libraries. One confound cannot be designed away: QUIC and CoAP over DTLS deliver different feature sets, so results have to be reported in relation to the functionality delivered.

5.2 Outlook II: Porting a Stack

The campaign presupposes a port that does not yet exist. The target adopted here is ngtcp2 with wolfSSL on RIOT [2]. Code size does not discriminate, since the platform pushes the complete system past the Class 2 budget whichever stack is chosen, so quicly's lack of compile-time feature switches (G3) costs little here. What binds is the data budget (G2) and whether a security-complete handshake is reachable at all (R2). On R2 the two picotls-based candidates share the minicrypto verification gap, which the wolfSSL binding avoids. On G2, ngtcp2 alone exposes a pluggable allocator, so `ngtcp2_mem` can be backed by a static pool without forking the library, where quicly's caller-supplied send buffer covers one buffer only. Two caveats qualify the choice. `ngtcp2_mem` governs where memory is taken from rather than how much is demanded, and wolfSSL is distributed under GPLv3 or a commercial licence, a constraint on closed firmware that R5 does not capture.

Five work items follow, each also a research question. *Memory discipline:* back `ngtcp2_mem` with static pools for one connection and few streams, and establish the minimum viable state representation for a single-connection endpoint — whether it fits below the roughly 23 KB quant pre-allocated without violating RFC 9002. *Cryptographic offload:* bind wolfSSL to the platform accelerator through PSA Crypto and quantify the share of Eggert's 5.10 s connection time attributable to software ECC. Should that share dominate, QUIC on constrained nodes is a problem of cryptographic integration rather than of transport design. *Certificate avoidance:* evaluate raw public keys [33] and external PSK against full X.509 chains, on which Saif and Matrawy [27] report 37.5% of CPU time, though in a virtual machine. *Small-MTU operation:* implement DPLPMTUD [8] and characterise QUIC as the effective MTU approaches the 6LoWPAN fragmentation regime, where the 1200-byte rule turns from an inconvenience into a design conflict. This may be the strongest argument for a constrained profile. *Sleep and connection lifetime:* determine whether 0-RTT resumption with tickets in flash beats holding a connection across the minutes or hours a duty-cycled device sleeps, for which QUIC's idle timeout, keep-alive and migration mechanisms were not designed.

The choice of stack should be tested before the campaign rather than argued. Cross-compiling the three transport cores with `-Os` and instrumenting peak al-

locator use for one connection would settle it empirically. Two predictions are offered for falsification. First, a security-complete ngtcp2 client with wolfSSL will need more RAM than quant's 34 KB, because quant could not verify peer signatures and its figure excluded the cryptographic libraries. Second, the spread between the three candidates will be smaller than the distance from any of them to the Class 2 budget, so that the platform rather than the choice of stack decides feasibility. Should the second hold, the port is worth undertaking for the evidence it produces rather than for the device class it reaches.

5.3 Open Questions

Whether QUIC should be adapted for constrained nodes at all is open, given that a complete EDHOC exchange needs less than a tenth of the bytes of QUIC's mandatory first datagram. If it should, the instrument is in question: a profile restricting implementation choices would address the divergence documented by Marx et al., a transport extension would not. And the unit of comparison is unsettled, since counting only bytes and joules undervalues the properties that motivated the reuse argument.

References

1. Alibaba: XQUIC — client and server QUIC library. https://github.com/alibaba/xquic, accessed 29 July 2026 (2026)
2. Baccelli, E., Gündoğan, C., Hahm, O., Kietzmann, P., Lenders, M.S., Petersen, H., Schleiser, K., Schmidt, T.C., Wählisch, M.: RIOT: An open source operating system for low-end embedded devices in the IoT. IEEE Internet of Things Journal **5**(6), 4428–4440 (2018). https://doi.org/10.1109/JIOT.2018.2815038
3. Bishop, M.: HTTP/3. RFC 9114, IETF (Jun 2022). https://doi.org/10.17487/RFC9114
4. Bormann, C., Ersue, M., Keränen, A.: Terminology for constrained-node networks. RFC 7228, IETF (May 2014). https://doi.org/10.17487/RFC7228
5. Eggert, L.: Towards securing the Internet of Things with QUIC. In: Proceedings of the Workshop on Decentralized IoT Systems and Security (DISS), NDSS Symposium. Internet Society (2020). https://doi.org/10.14722/diss.2020.23001
6. Eggert, L.: quant — QUIC transport protocol implementation. https://github.com/NTAP/quant, accessed 26 July 2026 (2026)
7. EMQ Technologies: Can ESP32 run MQTT over QUIC? we had a try. Company blog, https://www.emqx.com/en/blog/can-esp32-run-mqtt-over-quic, accessed 26 July 2026 (Sep 2025)
8. Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., Völker, T.: Packetization layer path MTU discovery for datagram transports. RFC 8899, IETF (Sep 2020). https://doi.org/10.17487/RFC8899
9. Fernández, F., Zverev, M., Garrido, P., Juárez, J.R., Bilbao, J., Agüero, R.: And QUIC meets IoT: performance assessment of MQTT over QUIC. In: 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob). IEEE (2020). https://doi.org/10.1109/WiMob50308.2020.9253384

10. H2O project: picotls — TLS 1.3 implementation in C. https://github.com/h2o/picotls, accessed 27 July 2026 (2026)
11. Huitema, C., contributors: picoquic — minimalist implementation of the QUIC protocol. https://github.com/private-octopus/picoquic, accessed 26 July 2026 (2026)
12. Iyengar, J., Swett, I.: QUIC loss detection and congestion control. RFC 9002, IETF (May 2021). https://doi.org/10.17487/RFC9002
13. Iyengar, J., Thomson, M.: QUIC: A UDP-based multiplexed and secure transport. RFC 9000, IETF (May 2021). https://doi.org/10.17487/RFC9000
14. Jaeger, B., Zirngibl, J., Kempf, M., Ploch, K., Carle, G.: QUIC on the highway: Evaluating performance on high-rate links. In: IFIP Networking Conference. pp. 1–9. IEEE (2023). https://doi.org/10.23919/IFIPNetworking57963.2023.10186365
15. Jeddou, S., Fernández, F., Diez, L., Baina, A., Abdallah, N., Agüero, R.: Delay and energy consumption of MQTT over QUIC: An empirical characterization using commercial-off-the-shelf devices. Sensors **22**(10), 3694 (2022). https://doi.org/10.3390/s22103694
16. Jung, J.H., Nam, H.B., Choi, D.K., Koh, S.J.: Use of QUIC for CoAP transport in IoT networks. Internet of Things **24**, 100905 (2023). https://doi.org/10.1016/j.iot.2023.100905
17. König, M., Rust, S., Zitterbart, M., Scheuermann, B.: Examining the heterogeneous throughput performance landscape of QUIC implementations. In: IFIP Networking Conference. IEEE (2025)
18. Kumar, P., Dezfouli, B.: Implementation and analysis of QUIC for MQTT. Computer Networks **150**, 28–45 (2019). https://doi.org/10.1016/j.comnet.2018.12.012
19. Marx, R., Herbots, J., Lamotte, W., Quax, P.: Same standards, different decisions: A study of QUIC and HTTP/3 implementation diversity. In: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ). pp. 14–20. ACM (2020). https://doi.org/10.1145/3405796.3405828
20. Mattsson, J.P., Palombini, F., Vučinić, M.: Comparison of CoAP security protocols. Internet-Draft draft-ietf-iotops-security-protocol-comparison-04, IETF (Mar 2024)
21. Montenegro, G., Kushalnagar, N., Hui, J., Culler, D.: Transmission of IPv6 packets over IEEE 802.15.4 networks. RFC 4944, IETF (Sep 2007). https://doi.org/10.17487/RFC4944
22. ngtcp2 project: ngtcp2 — QUIC protocol library. https://github.com/ngtcp2/ngtcp2, accessed 26 July 2026 (2026)
23. QUIC Working Group: Implementations and tools. https://quicwg.org/implementations, accessed 26 July 2026 (2026)
24. Rescorla, E.: The transport layer security (TLS) protocol version 1.3. RFC 8446, IETF (Aug 2018). https://doi.org/10.17487/RFC8446
25. Rescorla, E., Tschofenig, H., Modadugu, N.: The datagram transport layer security (DTLS) protocol version 1.3. RFC 9147, IETF (Apr 2022). https://doi.org/10.17487/RFC9147
26. RIOT-OS project: wolfSSL embedded SSL/TLS library — RIOT package. https://api.riot-os.org/group__pkg__wolfssl.html, accessed 27 July 2026 (2026)
27. Saif, D., Matrawy, A.: A pure HTTP/3 alternative to MQTT-over-QUIC in resource-constrained IoT. arXiv:2106.12684 (2021)
28. Selander, G., Mattsson, J.P., Palombini, F.: Ephemeral diffie-hellman over COSE (EDHOC). RFC 9528, IETF (Mar 2024). https://doi.org/10.17487/RFC9528

29. Selander, G., Mattsson, J.P., Palombini, F., Seitz, L.: Object security for constrained RESTful environments (OSCORE). RFC 8613, IETF (Jul 2019). https://doi.org/10.17487/RFC8613
30. Shelby, Z., Hartke, K., Bormann, C.: The constrained application protocol (CoAP). RFC 7252, IETF (Jun 2014). https://doi.org/10.17487/RFC7252
31. Thomson, M., Turner, S.: Using TLS to secure QUIC. RFC 9001, IETF (May 2021). https://doi.org/10.17487/RFC9001
32. wolfSSL Inc.: wolfSSL QUIC support. https://www.wolfssl.com/wolfssl-quic-support/, accessed 26 July 2026 (2026)
33. Wouters, P., Tschofenig, H., Gilmore, J., Weiler, S., Kivinen, T.: Using raw public keys in TLS/DTLS. RFC 7250, IETF (Jun 2014). https://doi.org/10.17487/RFC7250