Towards securing the Internet of Things with QUIC

Lars Eggert
Technical Director, Networking
2020-9-14
QUIC on IoT devices

Why? Reuse & leverage
Warpcore

- **Minimal, BSD-licensed, zero-copy UDP/IP/Eth stack**
- Meant to run on netmap, can use Socket API as fallback
- 3700 LoC (+ 3000 LoC w/netmap), C
- Exports generic zero-copy API
- **Device OS has LWIP = just works (after some patch submissions)**
- **RIOT has GNRC = needs own backend**
  - RIOT port of LWIP unfortunately broken
  - GNRC lacks key features (poll/select, IPv4, etc.)

Quant

- **QUIC transport stack** (i.e., no H3)
  - Focus: high-perf datacenter networking
  - Client and server modes
  - 10,300 LoC, C
- **Warpcore for UDP**, otherwise uses:
  - `khash` (from klib, modified)
  - `timing wheels` (Ahern’s timeout.c, modified)
  - `tree.h`
  - `bitset.h`
  - `picotsl`
  - `cifra`
  - `micro-ecc`
## System hardware and software

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

© 2020 NetApp, Inc. All rights reserved.
Measurements

Code and static data size
Build size: baseline

- Compiled code and static data size
  - **Application**
    - Argon app has more features, hence larger
  - **QUIC**
    - Already only uses single-precision FP
  - **TLS**
Eliminate costliest 64-bit math, i.e., division and modulus
- All are by constants, can multiply by magic number and right shift

Use 32-bit width
- for many internal variables, e.g.,
  - Packet numbers
  - Window sizes
  - RTT (μs)

Not fully spec-conformant, but unlikely to matter in practice for IoT
**Build sizes: client-only mode**

- **Disable server functionality**
- Unlikely to be of much use for IoT, esp. when battery-powered
- Also makes client use zero-length CIDs
- Large gain at the TLS layer!

- (Server-only mode: future work)
Build sizes: **minimally-required crypto**

- **Disable non-required crypto**, leaving
  - TLS_AES_128_GCM_SHA256 cipher suite
  - secp256r1 key exchange

- More gains at the TLS layer!

- **Could fully eliminate cifra & micro-ecc if HW crypto was accessible from OSs...**

- Together, reductions of 25-30% so far, without much loss in functionality

- Can save more by turning off functionality...
Build sizes: no migration

- **Connection migration** = switching an established connection to a new path
- Likely **unnecessary for IoT usage**
Build sizes: no error reasons

- QUIC allows plaintext “reason” strings in CONNECTION_CLOSE frames
- No protocol usage, only for human consumption
- Quant by default uses those heavily & verbosely
- So don’t
Build sizes: **no stateless resets**

- **Stateless reset** = signal to peer that local end has no more state for a connection
- To handle, need to be able to identify *which* connection RX’ed SR is for
- **Tradeoff**: handle SR vs. needlessly RTX
Build sizes: drop reordered 0-RTT

- Caching 0-RTT packets arriving out-of-order can avoid RTX
- Also has an overhead
- Tradeoff: cache vs. force RTX
Caching any out-of-order CRYPTO or STREAM data can avoid RTX

Also has an overhead

Tradeoff: cache vs. force RTX

Build sizes: drop all reordered data
Build sizes: don’t maintain connection info

- Quant maintains a TCP_INFO-like struct about each connection:

  pkt_in_valid = 40
  pkt_in_invalid = 0
  pkt_out = 10
  pkt_out_lost = 0
  pkt_out_rtx = 0
  rtt = 0.049 (min = 0.000, max = 0.087, var = 0.027)
  cwnd = 14840 (max = 14840)
  ssthresh = 0
  pto_cnt = 0

  frame                  code        out         in
  PADDING                0x00       2941       1214
  PING                   0x01          1          1
  ACK                    0x02          6           7
  CRYPTO                 0x06          3           5
  NEW_TOKEN              0x07          0           3
  STREAM                 0x08          1          29
  MAX_STREAM_DATA        0x11          1           0
  NEW_CONNECTION_ID      0x18          3           1
  RETIRE_CONNECTION_ID   0x19          1           2
  CONNECTION_CLOSE_APP   0x1d          1           1
  HANDSHAKE_DONE         0xe1          0           2
  strm_frms_in_seq = 33
  strm_frms_in_ooo = 1
  strm_frms_in_dup = 0
  strm_frms_in_ign = 0

- Don’t do that
Measurements

Stack and heap usage
Stack and heap usage

- **Instrumented binaries** to log stack and heap usage on function enter/exit
  - cifra and micro-cc *NOT* instrumented
    - Too many small functions, too much log data
  - Shown results are therefore **lower bounds**
    - Approximate the case *if* HW did crypto
- **Time units not shown** on purpose
  - Run takes tens of seconds due to 112.5Kb/s serial
- **Random 20%** of data points plotted to reduce overplotting

![Graphs showing stack size distribution](image)
Stack usage: *init phase*

- Quant and Warpcore initialization
- On ESP32, includes WLAN association = longer duration
- Minimal stack usage, few 100s of B
Stack usage: open phase

- Open connection to server
- **Public key crypto** as part of handshake
- **Stack usage peaks** at almost 3 KB
- **Not great** for IoT usage
  - 1 KB RIOT stack default
  - 6 KB Device OS stack default
- Optimizations needed
  - picotls uses stack-allocated buffers
Stack usage: transfer phase

- RX data from server
- Symmetric crypto
- Stack usage is lower at around 1 KB
- Still not super-great for IoT
- Optimizations needed
Stack usage: close phase

- Close connection with server and de-init
- **Stack usage dropping** down to initial values

- Overall, unfortunately, **peak stack usage** is what matters
Heap usage

- **Heap usage jumps** on allocation/deallocation of packet buffers
- 15 buffers @ 1500 B each

- Baseline heap usage on Argon much higher
  - DeviceOS executing in background
Heap usage

- During open phase, slight increase in heap
- Allocation of additional per-connection dynamic state
Heap usage

- **Flat heap usage**
  during transfer phase
- **Nice!**
Measurements

Energy and performance
Energy measurements

- Argon with 2000 mAh 3.7 V LiPo battery
- **Two runs** after full charges
  - Only 1-RTT connections
  - (Initial 1-RTT followed by) only 0-RTT connections
- Ran for ~2.5 days non-stop
  - 29,338 1-RTT connections (~0.90 J/conn)
  - 31,844 0-RTT connections (~0.83 J/conn)
- **Very preliminary!**
  - Argon-internal voltage reporting is coarse
  - Single run only
  - Hesitant to draw conclusions
Performance measurements

- Data from the same runs used for energy measurements
- Median 1-RTT connection took 5.10 s
- Median 0-RTT connection took 4.74 s
- Open questions
  - Why does 0-RTT show more of a slope?
  - Why is 1-RTT sometimes faster? (Loss?)
Future work

Lots and lots
Future work

Measurements

- Measure **data upload**
- **Vary parameters** of measurement
  - Object sizes, streams, connections, etc.
- **Compare** against other protocols
  - TCP, TLS/TCP, CoAP, MQTT, etc.
- **Compare** different IoT boards
- More accurate **energy measurements**

Implementation

- Add **H3** binding & measure
- Make **picotls** not use stack buffers
- **Better data structures** w/less heap churn
- Use **HW crypto** (performance & energy)
- **Drop 0-RTT** to shrink code size?
- IP over **BLE or 802.15.4** instead of WLAN
  - WLAN on ESP32 is 115 KB (45% of OS size)
- Can we scale down to **16-bit controllers**?
Thank you

Questions later?
lars@netapp.com