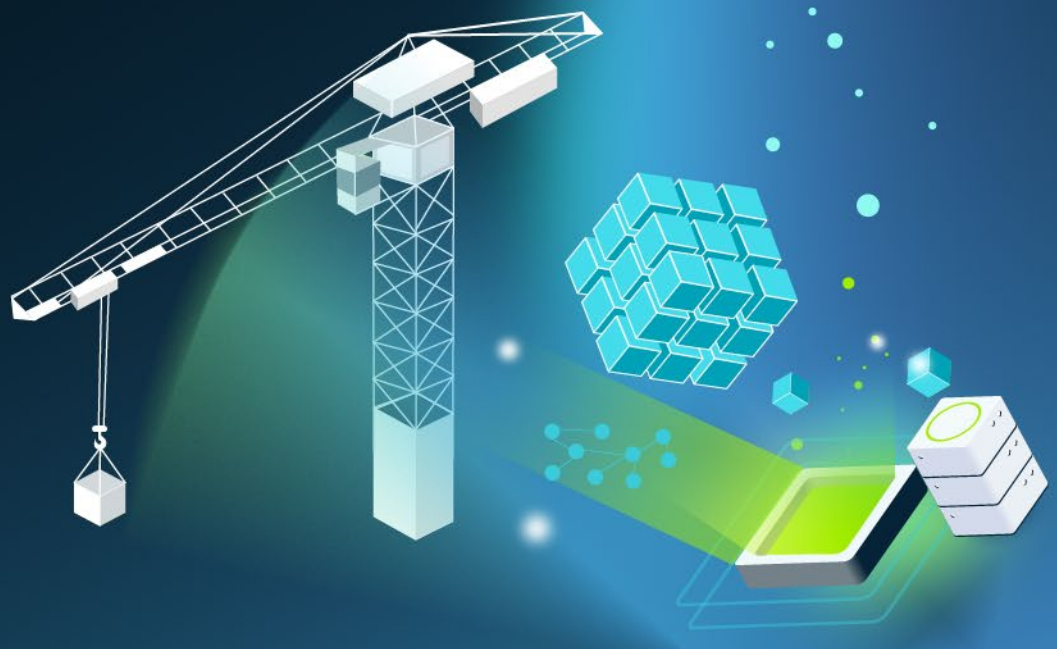
Hardware TCP offloading focused on the MQTT protocol

Efficient Connectivity: TCP Offloading for Enhanced Performance in MQTT-based communication

- Offloading protocols in SmartNICs
- Overview of available HW solutions
- SmartNIC-based ideas explored to speed up MQTT Publish packet distribution
    - TCP monitoring
    - Network stack integration with DPDK
    - Socket interception



codilime

Executive summary

Publication purpose

This publication provides:

- An overview of issues related to implementing partial TCP-based protocol offloads in SmartNICs,
- An outline of currently available hardware offloads and why they are not sufficient,
- A list and description of solutions explored by our CodiLime engineers.

Use case

The main goal of our activity is to tackle the problem of implementing a partial offload for TCP-based protocols, MQTT specifically, with a focus on MQTT Publish packet distribution. Reduction of latency in this area may be required in use cases where MQTT is used in time-critical applications, e.g. where the communication time frame is slim and/or the reaction to published data has to be rapid.

Solutions explored

- TCP monitoring - The offload sits between server and clients but is not a separate TCP endpoint. It creates two synchronization domains and translates SEQ/ACK numbers when passing packets between them.
- Network stack integration - The offload integrates its own network stack and acts as a separate TCP endpoint, listening on the same port as the server.
- Socket interception - The offload provides an API, allowing the host system to utilize the offload's network stack.

The outcome

Even with the most crude solution (i.e. TCP monitoring), a significant reduction in MQTT Publish packet distribution time was observed. From the end user's perspective, the solutions we have explored, although software-based in nature, may be classified as hardware offloads since they run on an add-in card, a SmartNIC. Therefore they address some of the issues identified in pure hardware solutions (e.g. TOE) such as security, extensibility or resource limitations, making them an interesting area for further exploration and possible adoption.

What is MQ Telemetry Transport (MQTT)? **4**

Goal **5**

Offloading protocols in SmartNICs 6

Layer guarantees 6

Traffic filtering 6

Ethernet 6

IP 7

TCP 7

Overview of available HW solutions **13**

Ideas explored for MQTT Publish packet distribution **13**

TCP monitoring **14**

Network stack integration **19**

Design 19

Implementation 19

Problems addressed 20

Results 21

Lessons learned & next steps 21

Socket interception **22**

Design 22

Summary **25**

About the authors **26**

About CodiLime **27**

What is MQ Telemetry Transport (MQTT)?

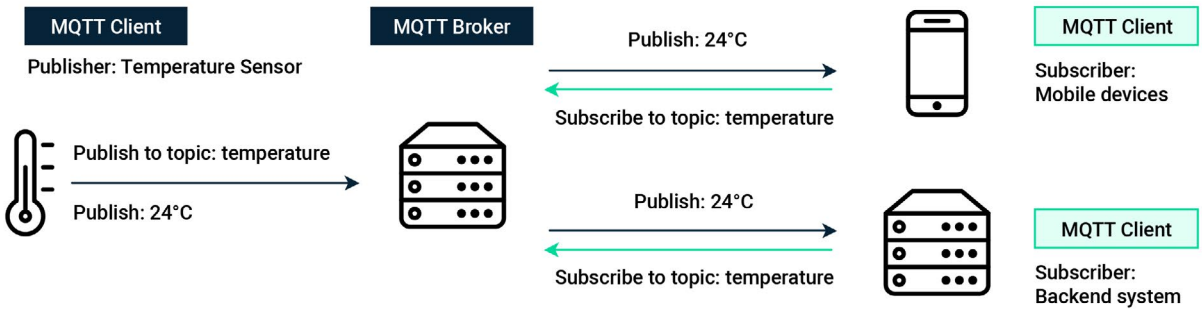
MQ Telemetry Transport (MQTT) is an OASIS-standard messaging protocol for IoT. By design, it is a lightweight, publish/subscribe messaging transport aiming at supporting large-scale deployments with storage and network bandwidth constraints.

MQTT is built on top of the TCP protocol and relies on it for data transmission. The MQTT-SN specification allows it to also run over UDP or Bluetooth.

Deployments consist of the following actors:

- **Broker** - server; request handler and message distribution center.
- **Publishers** - client devices that transmit messages to the broker. Messages are accompanied by a topic.
- **Subscribers** - client devices that communicate with the broker to subscribe to certain message topics by supplying one or more topic filters.

Upon message arrival, the MQTT broker matches the topic to topic filters and distributes the message to all applicable subscribers.



There are use cases where minimizing the time between message transmission and its successful distribution is crucial, making it a viable subject for hardware offloading:

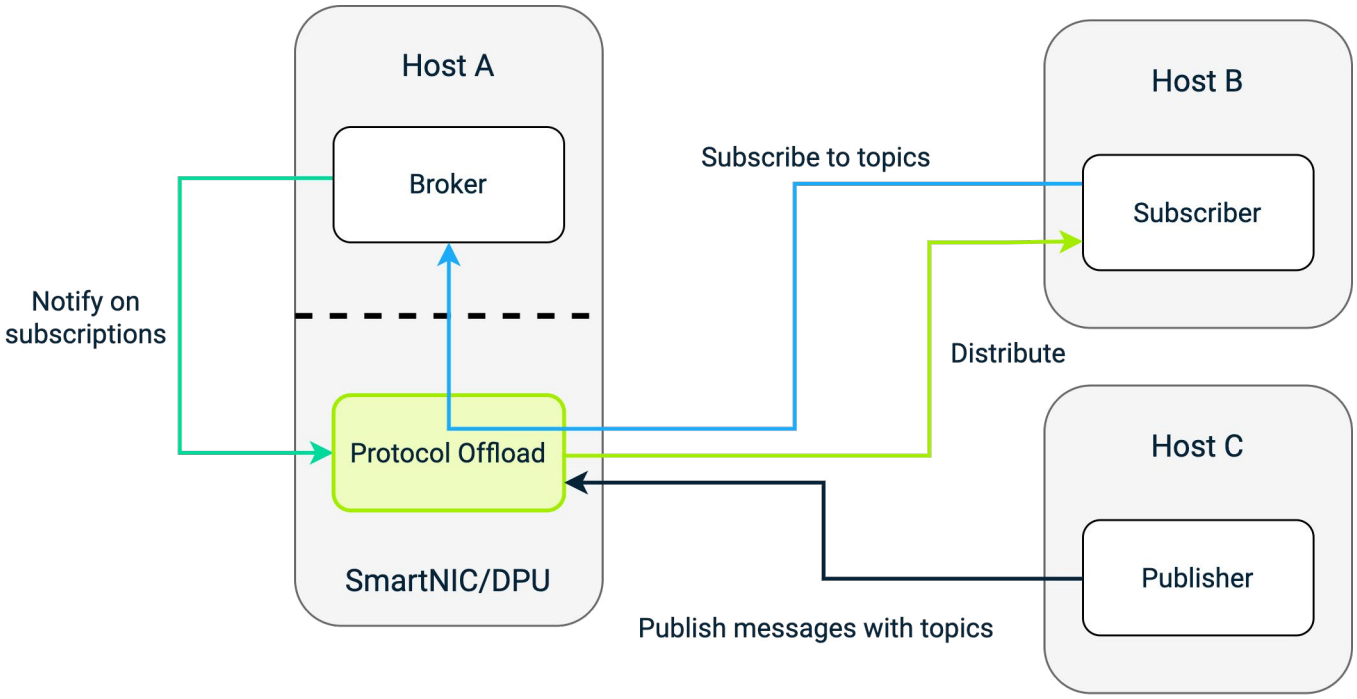
1. Deployments such as Internet-of-Vehicles, where fast-moving actors can establish a stable wireless connection for a very short period of time until they get out of range.

2. Solutions where immediate reaction to telemetry data is required. A good example is Matternet, where MQTT is used to monitor drone flight status in real-time.

Goal

The main goal of our activity is to tackle the problem of implementing a partial offload for TCP-based protocols. In contrast to packet filtering, for example, such an offload will need to act as an active endpoint - i.e. generate and process protocol-related packets independently from the host application - which poses additional challenges.Furthermore, a partial offload introduces more complexity which would not occur if we simply migrate the application responsible for handling the protocol from the host to the SmartNIC Arm cores.

When it comes to protocol choice, we have decided to focus on MQTT due to its popularity and simplicity. As for what specific aspect of that protocol will be offloaded, the most resource-consuming operation carried out by the MQTT Broker is the processing and distribution of MQTT Publish packets, where CPU cycle and memory usage increase with each extra client, topic and distribution.



Requirement-wise:

1. The offload will be responsible for capturing MQTT Publish packets coming from publishers. Those packets shall not be passed through to the broker.

2. The offload shall match MQTT Publish topic to relevant subscribers. In order to do so, the offload shall maintain a map of active subscriber-topic relationships.

3. The offload shall pass all the other traffic (such as MQTT Subscribe packets) to the broker.

4. The broker shall notify the offload of any changes to subscriptions which shall trigger an update to the subscriber-topic map.

Offloading protocols in SmartNICs

Layer guarantees

As we have mentioned before, the MQTT protocol offload needs to be an active endpoint; i.e. generate and process packets on behalf of the broker. The MQTT broker application running on the host relies on the OS network stack to provide features and guarantees for the TCP and lower layers via the socket API. Just like the broker, the offload will need to rely on the same guarantees to do its job. Unless the offload is relieved by a network stack or other mechanisms, it will need to handle the TCP and lower layers on its own.

Without going into detailed specifications for each layer the MQTT is based on, let's focus on the major aspects the offload will be dealing with.

Traffic filtering

Any offload will perform better if it deals with packets relevant only to the use case at hand (MQTT) and not the entire flow of traffic between the server and the clients. Most modern network adapters allow for some form of traffic filtering, whether it be flow-/packet-based mechanisms, dedicated application queues (Intel ADq), or a fully programmable P4 engine.

Ethernet

Mapping packets to physical ports

Packet transmission at the Ethernet layer requires that the offload correctly matches the destination MAC address to its physical ports. Depending on the dynamics and complexity of the MQTT infrastructure, the offload may be required to act as a:

- **Learning switch**, by listening in on incoming Ethernet packets, matching the source MAC to a port to build and maintain a forwarding table,
- **Static** forwarder, where a forwarding table is supplied from outside,
- **Hub**, by passing a packet from one port to all the other physical ports the SmartNIC has, including the abstract one towards the host system. If the offload is dealing with only two ports, it simply moves packets from one port to the other.

This requirement applies both to packets forwarded between client and server and packets generated independently by the offload.

IP

Fragmentation

As per RFC 791, endpoints can transmit IPv4 packets as big as 64kB, larger than Ethernet MTU in most cases. Such packets may be generated by network stacks combining multiple upper-layer payloads (see Nagle's algorithm) and will be split across multiple Ethernet frames. Due to the lossy nature of Ethernet, those frames may arrive out of order. Network stacks store the out of order fragments until they receive the rest of the IPv4 packet.

Error checking

Since the offload will generate and terminate IPv4 packets, it will be responsible for calculating and validating the IPv4 header checksum. If a given SmartNIC supports HW offloads for checksum calculation/validation, it may be used to relieve the offload.

Tx - Mapping IP to MAC addresses

Just as in the case of matching MAC addresses to physical ports, the offload can either relate the source IP to source MAC addresses at runtime or obtain the mapping from outside. Implementing the former option may become more complex when the offload has not recorded any packets between the server and the client but needs to generate and transmit packets (e.g. it also initializes its own TCP connections). Additionally, if the clients do not belong to a local network, the offload will need to direct traffic to the appropriate gateway. In such cases, the offload may need to support the ARP protocol.

Rx - Servicing the host-side server only

Packets arriving at the SmartNIC may be related to any server, which may cause the offload to service all of them, not just the one running on the SmartNIC's host system. This problem can be solved by filtering out the traffic by the server's destination IP address either in the SmartNIC before or inside the offload. The latter requires that the offload is supplied with the server's IP address during initialization.

TCP

Statefulness

Figure 6 from RFC 793 outlines a TCP connection state machine along with the allowed state transitions. The offload needs to keep track of the states of all connections related to the offloaded protocol to ensure that none of the packets it generates and transmits accidentally causes any connections to be closed.

Data ordering

TCP ensures that data streams are received by the upper layer in the same exact order in which they were originally sent. What this means is that the TCP layer always provides the receiving end with the next contiguous block of data, with no gaps in between. In case such a gap occurs in the data stream due to the loss of one of the packets, any data that belongs to the space past that gap will be buffered until the missing blocks are received and stream continuity is restored.

Reliability

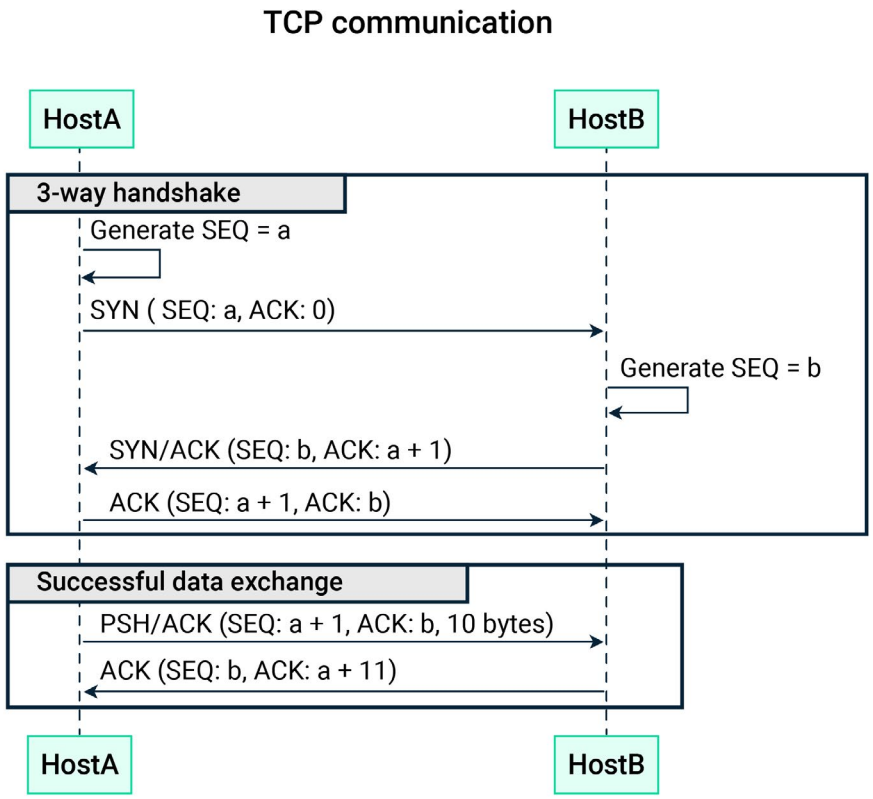
The TCP protocol is designed to provide reliability of data transfer between endpoints on an otherwise lossy and unreliable network. To ensure that each byte sent was successfully received, TCP endpoints exchange two numbers that accompany each transmitted packet:

- **SEQ** - starting offset in the connection data stream to which the packet data belongs,
- **ACK** - offset of the last received byte.

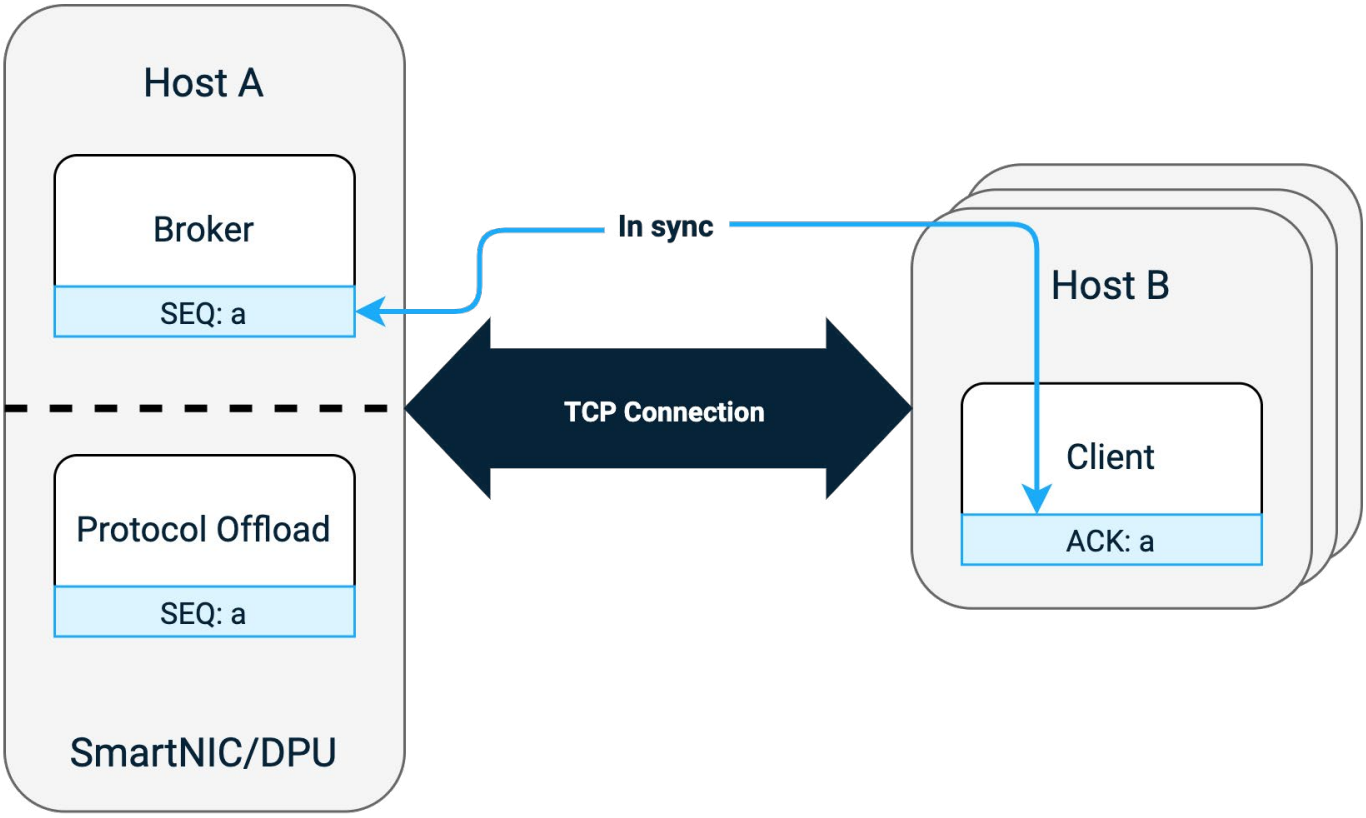
Establishing a TCP connection always begins with a three-way handshake to exchange initial SEQ/ACK values. Later on, each endpoint advances its SEQ/ACK value for each byte sent/received. Analysis of SEQ and ACK numbers allows an endpoint to figure out, whether:

a. The transmitted data was successfully received by the peer,

b. The incoming data was already received and is a duplicate.

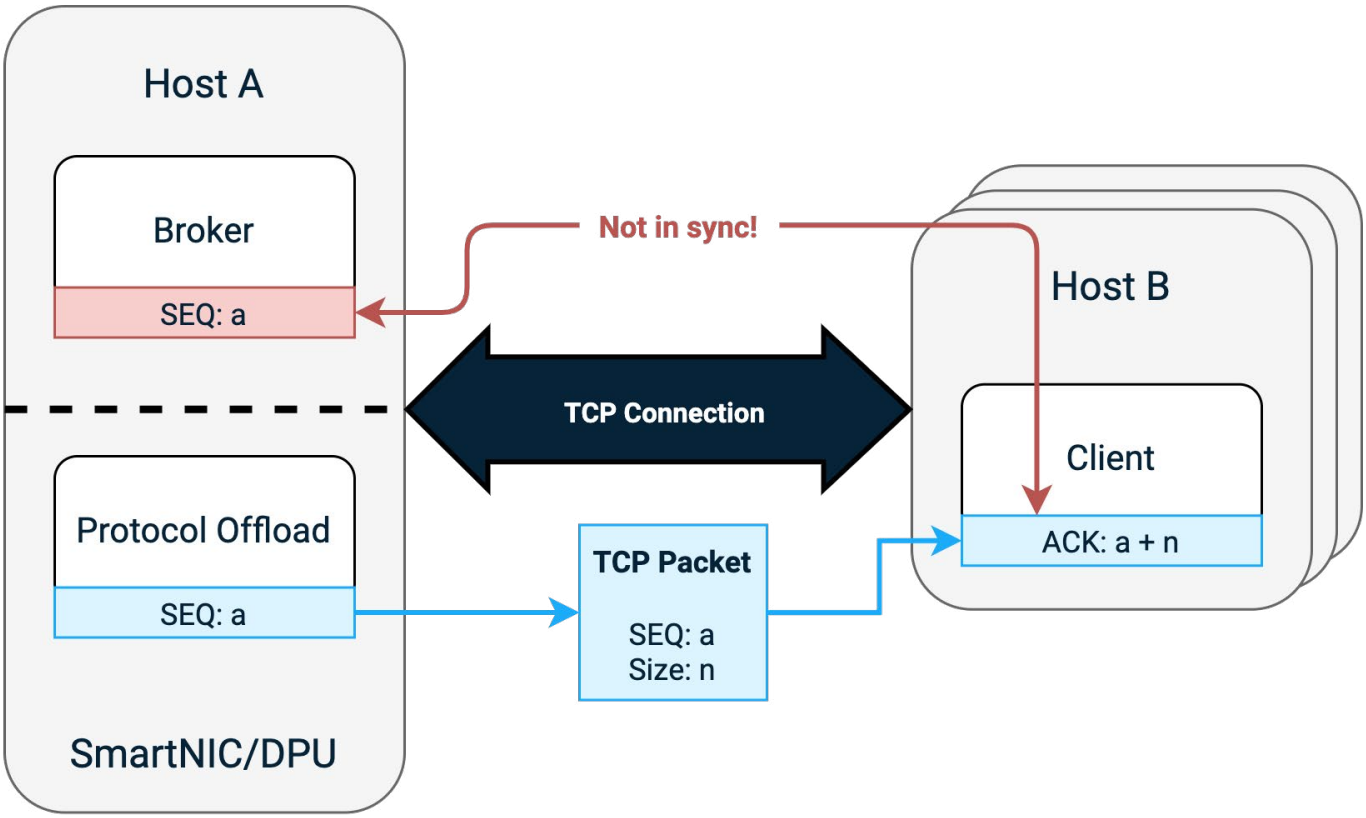


TCP was designed to be used for point-to-point communication between just two endpoints. Since the offload will be impersonating one of the endpoints (the server), it will need to use the same SEQ/ACK numbers as the server when generating packets targeting the clients. Otherwise, the client will completely drop the packets originating from the offload.



SEQ/ACK desynchronization

Packet generation and transmission from the offload to the clients shall be completely transparent to the server. However, this means that once the offload transmits a single valid TCP packet towards the client, the server and client views on the connection (SEQ/ACK numbers) are no longer synchronized. From that point going forward, all packets coming from the server will be dropped by the client. Lack of response from the client will cause the server to terminate the connection.



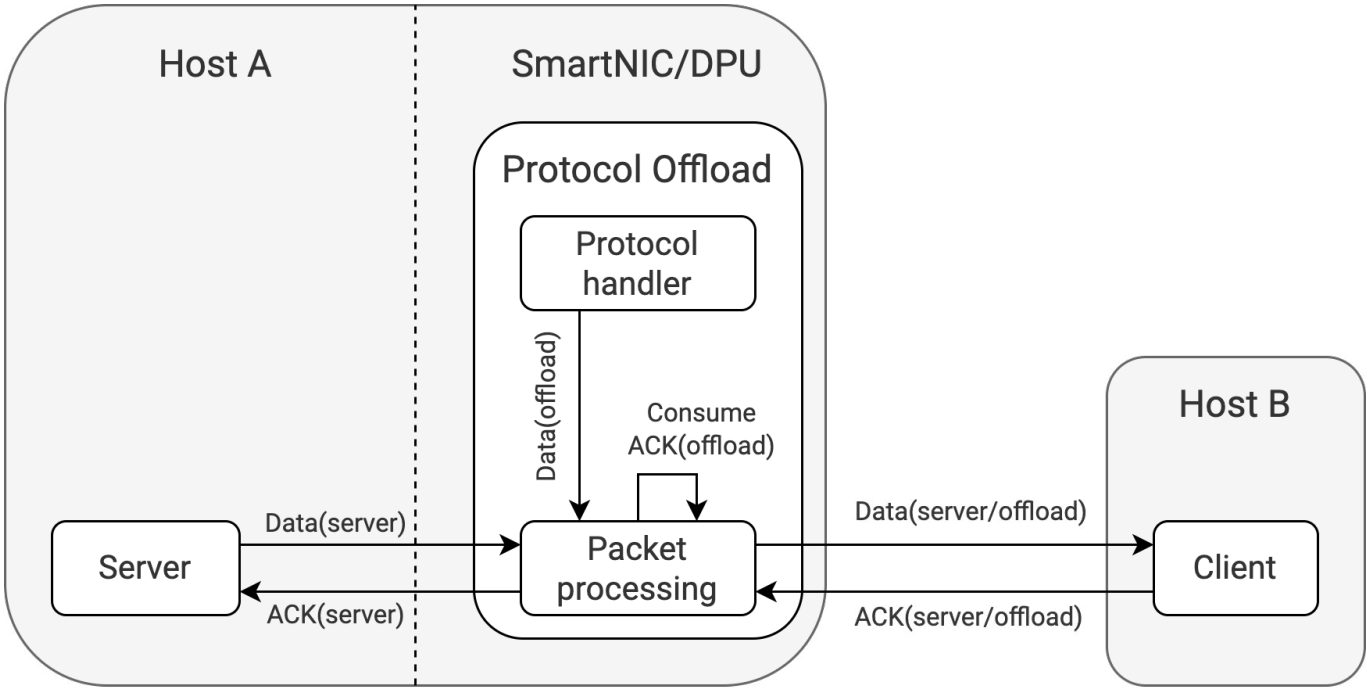
One way or the other, the offload will need to keep the server and clients convinced that their connections are healthy, despite the fact that their SEQ/ACK numbers are out of sync.

ACK matching

On the server host side, packet data is generated by either the server or the offload, but from the client's point of view, is transmitted only by a single TCP endpoint. Each data origin will require its data to be acknowledged. However, packet data origin is transparent to the client; it simply receives and acknowledges a continuous data stream. As a result, the ACKs received by the offload will be devoid of information about which entity generated the data. In order to know which data origin is being acknowledged by an incoming ACK packet, the offload will need to retain a map of data packets to their data origin and match the ACK SEQ/ACK numbers against that map. Packets originating from the offload can be immediately consumed. However, when it comes to acknowledging the server, the offload can take one of the two approaches:

1. When the offload receives a data packet from the server, it immediately responds with an ACK on behalf of the client. Later ACKs coming from the client on packets passed from the server can be dropped by the offload. This approach may seem to be easier to implement since the offload will need to match the incoming ACKs only to the data it generated. However, the absence of any ACKs from the client on the data generated by the server will require that data to be retransmitted by the offload.

2. The offload does not acknowledge server data packets but stores their SEQ numbers and length. When it matches the ACKs from the client to the data originating from the server, it passes those ACKs to the server.



Delayed ACK

One of the techniques used by contemporary network stacks to improve network performance is transmitting a single acknowledgement packet for more than a single data packet. This complicates the problem of ACK matching as the data being acknowledged by a single ACK packet may have been generated by both the server and the offload. To correctly deal with this problem, the offload will need to dissect a delayed ACK packet and match one or more packets to their origins and then apply one of the two approaches described in the previous section.

Although cumbersome, this problem can probably be alleviated by making sure that the clients configure their sockets with the **TCP_QUICKACK** option – see here.

Simultaneous transmission

Packets heading towards the client are generated by the server and the offload in an asynchronous manner. Even in the most optimistic case where those entities are in perfect synchronization in regards to SEQ/ACK numbers used, there is a risk that each of them generates and transmits a packet with the same SEQ number, causing the client to receive one of those packets and ignore the other. Since all of the packets coming from the server are first captured by the offload, the latter will be responsible for taking necessary actions to form a consistent data stream for the client to consume, e.g. put all data coming from different sources into a single FIFO queue and then send data from the queue assigning subsequent SEQ numbers to the outgoing packets.

Retransmissions

Since the offload is an active endpoint, in case an ACK is not received for the data it has generated, it is responsible for issuing retransmissions to ensure that data has been successfully delivered to the client.

Nagle's algorithm

According to RFC 896, a sender may pack multiple smaller messages into a single TCP payload to reduce protocol impact on the network bandwidth. Hence, a single client can issue a single Ethernet frame with a number of TCP-based protocol messages and expect an ACK packet in return. Some of those messages may need to be handled by the offload while the rest needs to be formed back into a valid packet and sent to the server for processing. Additionally, the ACK for the client must be formed of partial ACKs from each of the entities handling messages. Examples of how the offload can handle this problem are:

1. Immediately ACK the client. If some of the messages were passed to the server and the latter did not acknowledge them, the offload will need to retransmit the data on behalf of the client.

2. Keep track of data that needs to be acknowledged by the server. The offload may wait for ACKs from the server and send a single ACK to the client or use a SACK (selective ACK) to acknowledge each message

Although cumbersome, this problem can be alleviated by making sure that the clients configure their sockets with the TCP_NODELAY option – see here.

Congestion control

As per the end-to-end principle, TCP endpoints take active part in controlling traffic congestion. Assuming the offload does not establish TCP connections but operates on them, aside from manipulating SEQ/ACK numbers it may also need to monitor and adjust TCP header fields related to congestion control algorithms negotiated between the broker and the clients. This ebook will not go into the details of any of the congestion control mechanisms as each one requires a case study on its own.

Keep-alive

In rare cases where the offload captures and handles most of the traffic that would otherwise occur between the server and the clients and any of those endpoints expects the connection to be explicitly kept alive, the offload may be required to generate or handle TCP keep alive packets to prevent the existing connections from being terminated.

Overview of available HW solutions

Some of the problems and aspects of L2-L4 layers described in the previous section are relevant to regular OS network stacks and can be accelerated by hardware solutions currently existing on the market. The same mechanisms can be leveraged by a TCP-based protocol offload.

Packet/flow-based filtering allows for sifting the traffic to ensure that the offload is processing packets that are related only to the selected protocol; in our case: MQTT. Fewer packets received means fewer resources used and modern NICs may help deal with this problem by offering programmable pipelines as well as P4 engines.

Problems on the receiving side related to packet fragmentation and TCP data stream continuity can be taken care of by **large receive offload (LRO)** or **general receive offload (GRO)**, which collect and aggregate incoming smaller packets into a larger one before passing them to the upper layer.

In cases where the offload needs to pass a message that exceeds the TCP maximum segment side, it can utilize **TCP segmentation offload (TSO)** or **generic segmentation offload (GSO)** to split that message into fragments, prepend them with necessary headers and transmit them to the target endpoint.

Finally, a **TCP offload engine (TOE)** can be used to provide a TCP-based protocol offload with a fully-fledged TCP/IP network stack functionality. This means that all of the issues related to L2-L4 layers described in the previous section would have been dealt with. However, the adoption of hardware-based TOEs has still to occur, mainly due to the kernel community concerns around:

- security,
- updateability,
- hardware resource limitations,
- lack of flexibility,
- Rapid obsolescence.

There have been several attempts at addressing these concerns by realizing TOE in software running on SmartNICs (TASNIC, FlexTOE). Although potentially useful, these specific implementations are out of the scope of this ebook as they target Netronome Agilio CX cards. In our proof-of-concept efforts, we were dealing only with NVIDIA BlueField 2 DPUs.

Ideas explored for MQTT Publish packet distribution

The following sections describe proof-of-concept solutions we have tried in pursuit of a usable and efficient MQTT Publish packet distribution offload. The solutions are presented in chronological order, outlining our struggles and lessons learned as we went through them.

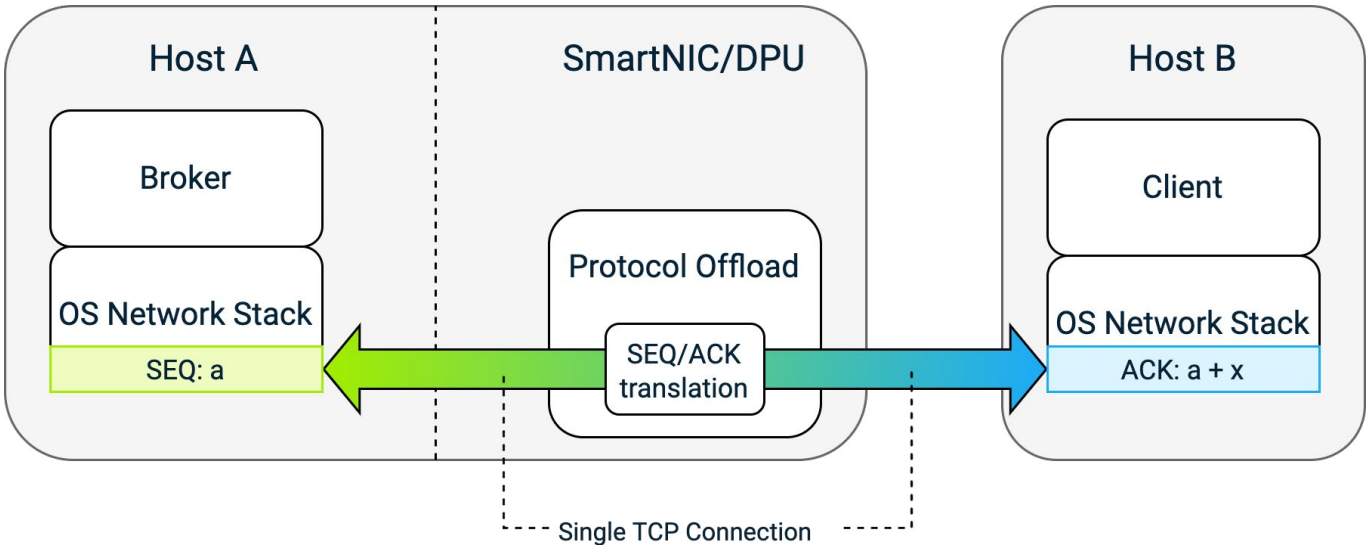
TCP monitoring

Our initial goal was to implement the simplest possible offload that would allow us to run a minimal scenario and evaluate at an early stage whether any further investment was viable.

Design

Conceptually the offload is placed on a network path between separate hosts running MQTT broker and MQTT clients. It is done by embedding specialized hardware - a SmartNIC - into the host running the broker. From the transport layer's perspective, it is not a TCP endpoint that establishes connections to any of the MQTT entities, but it monitors and operates on existing connections, processes, generates and transmits packets on behalf of connection endpoints.

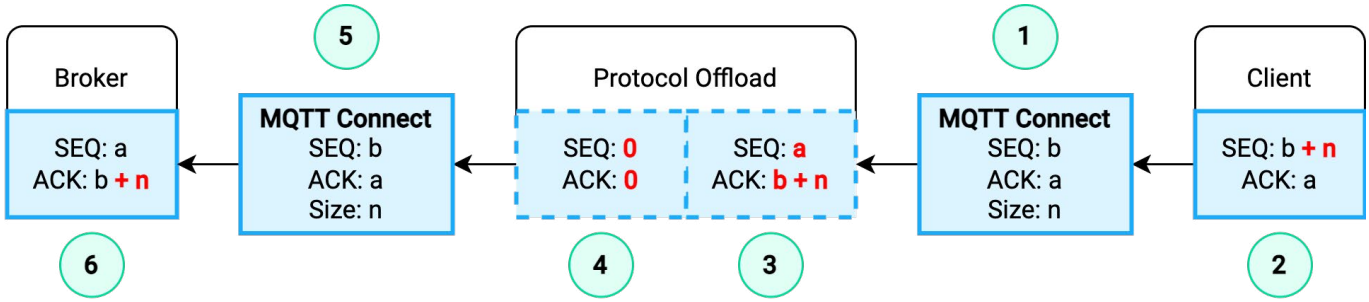
The main issue that we had to address so that the scenario would not immediately fail was TCP SEQ/ACK desynchronization. As described above, after a single packet has been generated and transmitted by the offload on behalf of the broker, the broker and the client will have a different view of the connection state; they will expect different SEQ/ACK numbers in the packets they will exchange. The general idea was to split the synchronization domain (the whole connection) into two local domains (broker-offload, client-offload) and introduce a SEQ/ACK translation mechanism to the offload. Packets forwarded from the broker to a given client would have their SEQ/ACK numbers adjusted to the values expected by the client, and vice versa. As a result, both sides will remain convinced that the packets they receive belong to their connection and accept them.



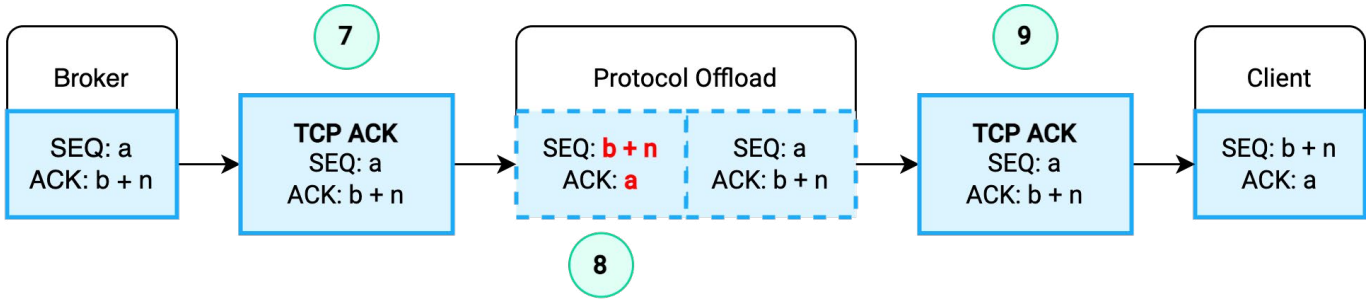
Endpoint peer abstraction

The main building block of the SEQ/ACK translation mechanism is **endpoint peer abstraction**. For each new connection, the offload instantiates two data structures called endpoint peers that represent how a given endpoint perceives the state of its peer, with a focus on SEQ and ACK numbers. These numbers are modified each time the offload transmits or receives a packet on behalf of a given endpoint. Let's analyze the mechanism step by step to better understand it.

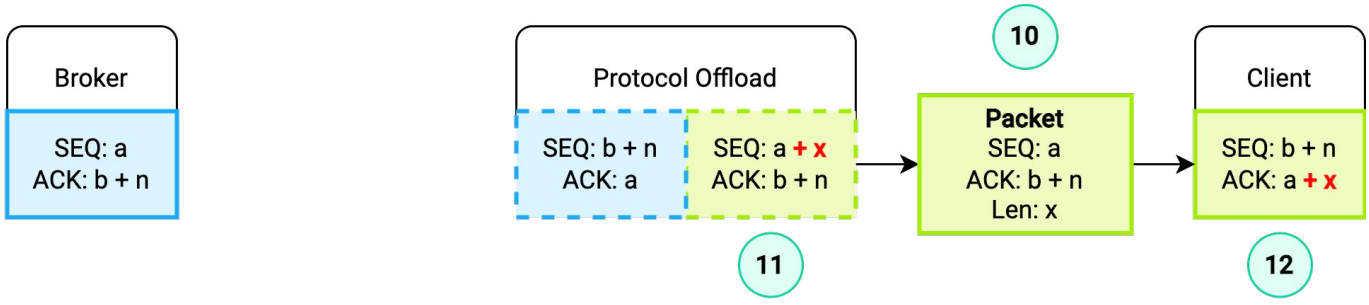
Let's assume that at a given point in time the broker and the client are in sync in regards to SEQ and ACK numbers with no packets in transit and the next transmission is an MQTT Connect packet sent by the client towards the broker (1), advancing the client's SEQ number (2). The offload intercepts that packet and sees it as a trigger to start monitoring a given TCP connection. It creates an endpoint peer structure with mirrored SEQ and ACK numbers extracted from the packet (3). ACK is also updated by the size of the received packet as if the abstract peer (broker) has received it. Then, the offload creates an uninitialized abstract peer (client) for the broker side (4). Finally, it passes the unmodified packet through (5) which increments the broker's ACK number (6).



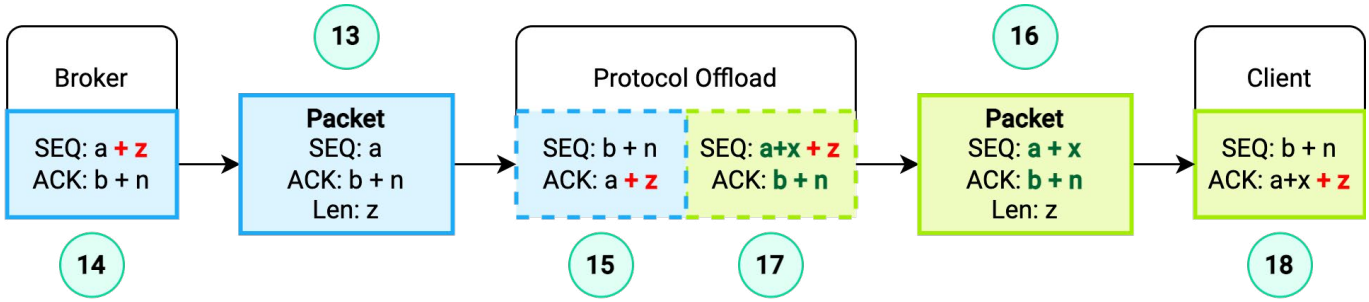
In response, the broker sends the ACK packet (7), which causes the offload to initialize its endpoint peer structure (8) and pass the unchanged packet to the client (9). There is no magic happening yet, until the connection becomes desynchronized.



At some point the offload will generate and transmit a packet to the client on behalf of the broker (10), incrementing the SEQ number of the endpoint peer representing the broker (11). On arrival, the ACK number of the client also advances (12).



While the TCP connection is now desynchronized, the green and blue domains are in sync. In order to allow the packets to pass between those domains, incoming packets have to be adjusted. Let's assume that what happens next is that the broker issues a packet that should be handled by the client (13) and increments its own SEQ (14). The offload intercepts the packet and updates the client endpoint peer's ACK (15). Next, the offload substitutes the packet's (16) SEQ and ACK fields with the numbers from the broker endpoint peer and transmits that packet while also updating the broker endpoint's peer SEQ (17). Finally, the packet is received by the client which increments its ACK (18).

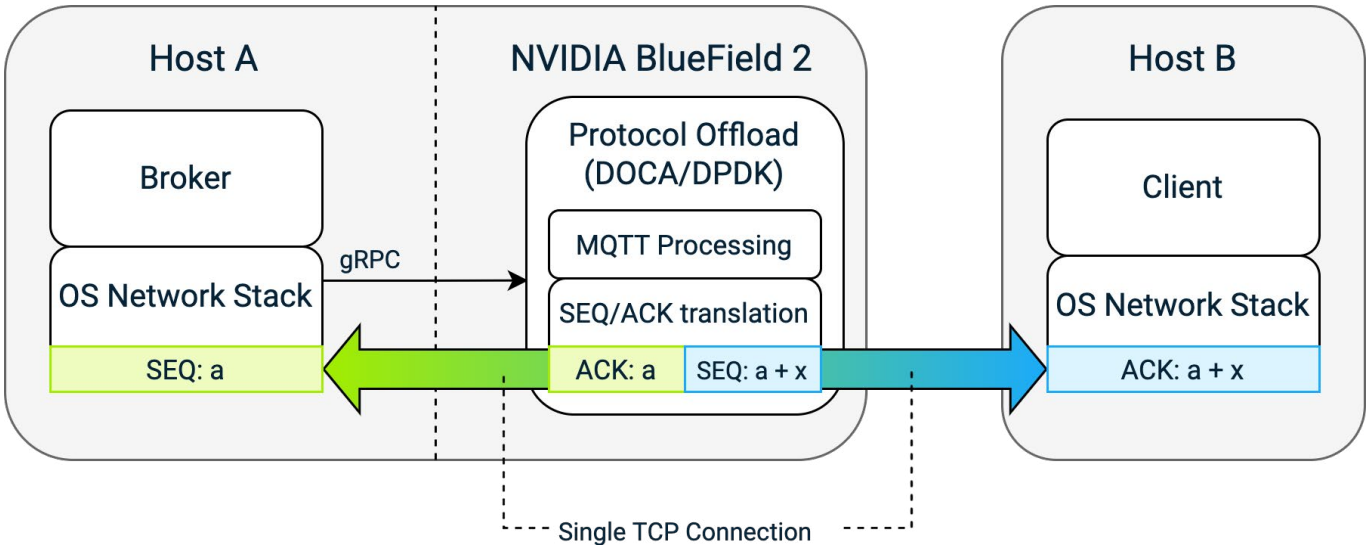


Any further communication happening between the broker, the offload and the clients takes place based on the mechanisms described above.

Implementation & deployment

We have decided to implement the offload as a DOCA/DPDK software application and run it on the Arm cores of NVIDIA BlueField 2 DPU. Application threads receive all the packets from two network interfaces: one facing the broker, the second facing Host B which runs all the MQTT clients. Protocol offload runs a gRPC server that is notified by the broker whenever a new subscription is registered.

The rest of the requirements this application implements are covered in the Goal section.



Problems addressed

The table below outlines how many TCP protocol features, guarantees and issues are addressed by our MQTT protocol offload.

Problem	Solved?	Details
SEQ/ACK desynchronization	Yes	Simple monitoring & translation
ACK matching	No	Offload always forwards ACK to broker
Simultaneous transmit	Yes	Single core doing all Tx/Rx ops
Retransmissions	No	Offload does not retransmit packets Broker may choose to do so for its packets
Delayed ACK	No	Offload does not analyze a delayed ACK
Nagle's algorithm	No	Offload does not split TCP payload Offload does not buffer partial TCP payloads
Keep-alive	No	Offload does not send TCP keep-alives Broker may choose to do so
Congestion control	No	Offload does not employ any congestion control algorithms

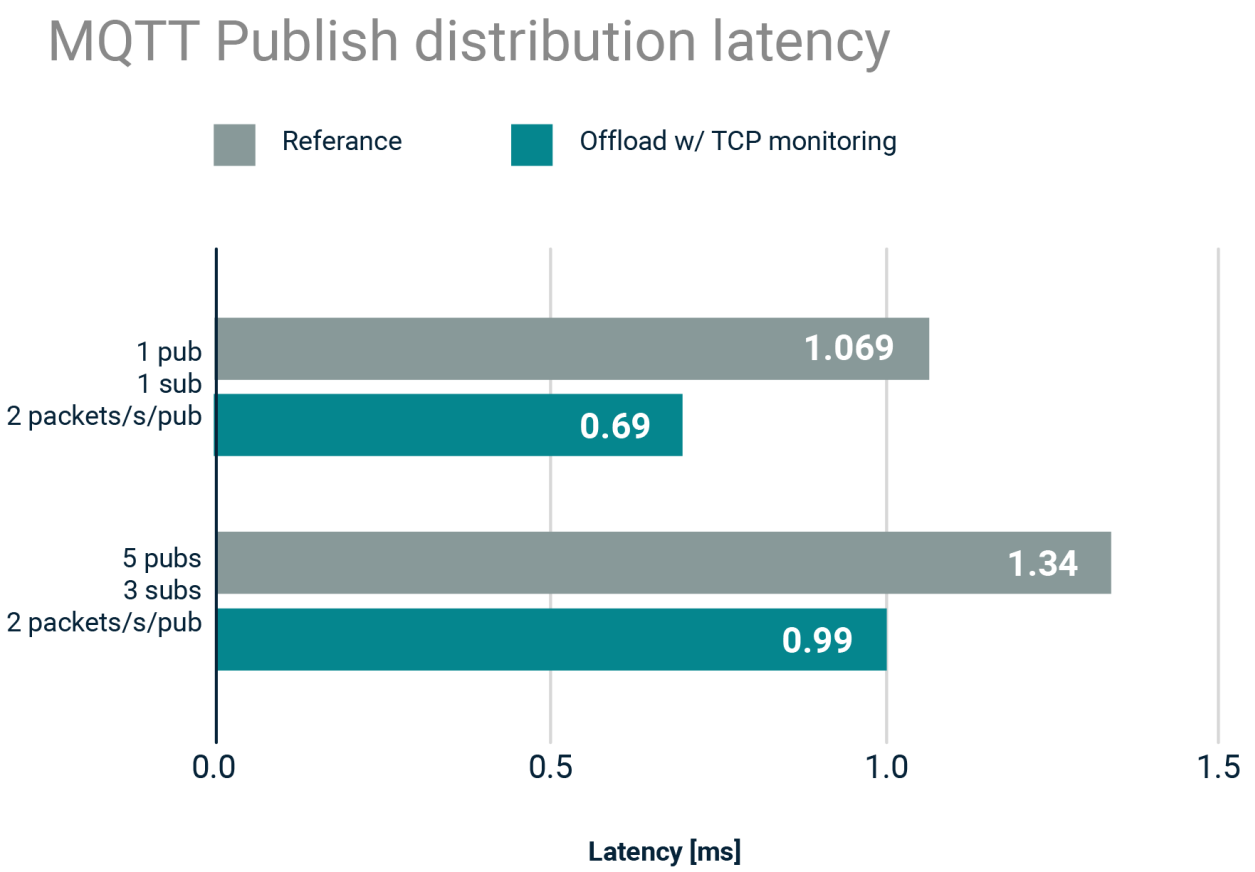
While most of them are not handled, tackling just the SEQ/ACK desynchronization allowed us to run a minimal scenario and gather some results.

Test results

We have used the JMeter testing harness with MQTT Plugin to:

- Spawn a given number of clients of each type and packets issued by each publisher every second.
- Measure the time between issuing an MQTT Publish packet and arrival of that packet to the last subscriber waiting for it.

The results are impacted by a significant, yet constant, overhead introduced by JMeter's MQTT plugin. Nevertheless, they show a significant reduction of the MQTT Publish distribution time when our protocol offload is operational compared to the reference scenario where the broker and clients communicate directly.



One thing that is noticeable is that the packet load defined for our application is vestigial in both testing configurations. Increasing the number of publishers or subscribers caused the system hosting the clients to squash multiple packets into a single TCP payload (Nagle's algorithm), cutting some of those packets down the middle. Our protocol offload was not ready to handle such packets. Additionally, when the offload was busy and did not respond to a given publisher with an ACK within a ~250 ms time frame, it started receiving retransmissions which broke the simple SEQ/ACK tracking mechanism.

Lessons learned & next steps

Overall, the results we have gathered looked promising enough to justify further tampering with the protocol offload. What was expected is that leaving most of the TCP guarantees unaddressed caused the offload to break under any significant load. Additionally, the concept of endpoint peer abstraction seemed to be very similar to a socket implementation, apart from the fact that they did not constitute a regular TCP endpoint that would issue or accept connections. That is where the second design came into play.

Network stack integration

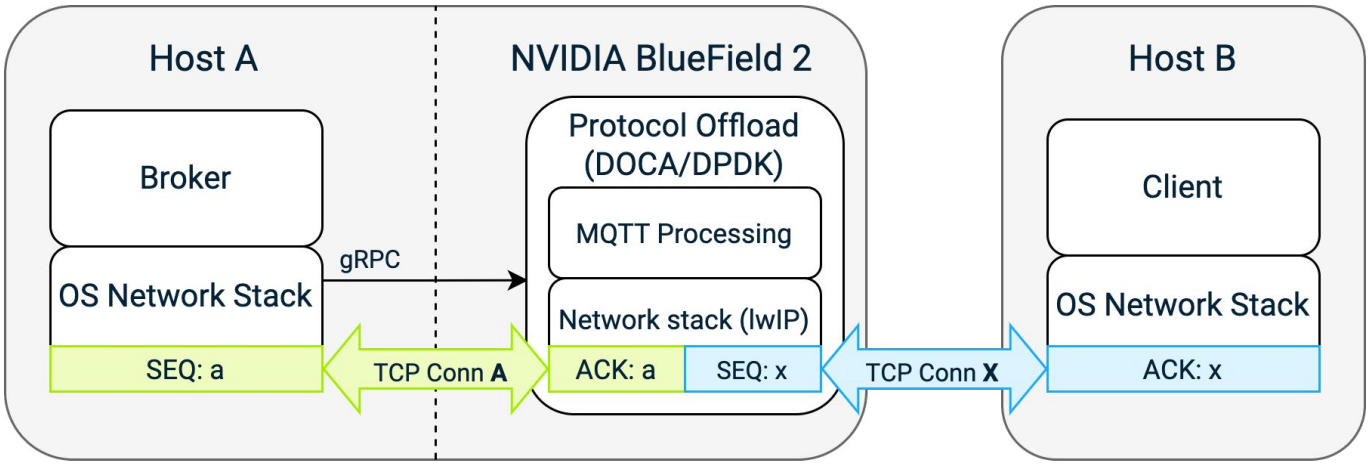
Instead of trying to tackle the problems related to offloading TCP-based protocols one by one, we have decided to integrate a fully-fledged network stack into our DOCA/DPDK application. This way we should address all of these problems and allow us to measure the performance impact of introducing a more complex packet processing mechanism.

Design

The protocol offload evolves from working on top of an already established TCP connection and translating packets back and forth to a situation where the offload itself becomes an independent endpoint with its own IP address to which clients will connect to. Once the offload accepts an incoming connection, it establishes a peer connection to the broker using the same TCP port as the client has used. There are still two synchronization domains but they are managed by mechanisms compliant with the TCP RFC.

Implementation

The main decision to make was the choice of a network stack for integration with our offload. The dpdk.org ecosystem page outlines a few projects that we could utilize: ANS, F-Stack, mTCP or TLDK. However, all of them are based on older DPDK versions than the one integrated into the DOCA SDK we were using. Early assessment showed that the effort required to apply the necessary adjustments to those projects was too big for us to pursue - it is a proof of concept after all. We ended up integrating the lwIP network stack, which is designed specifically for embedded systems. It also had a ready port for Unix systems and seemed easy to adjust to using DPDK APIs.



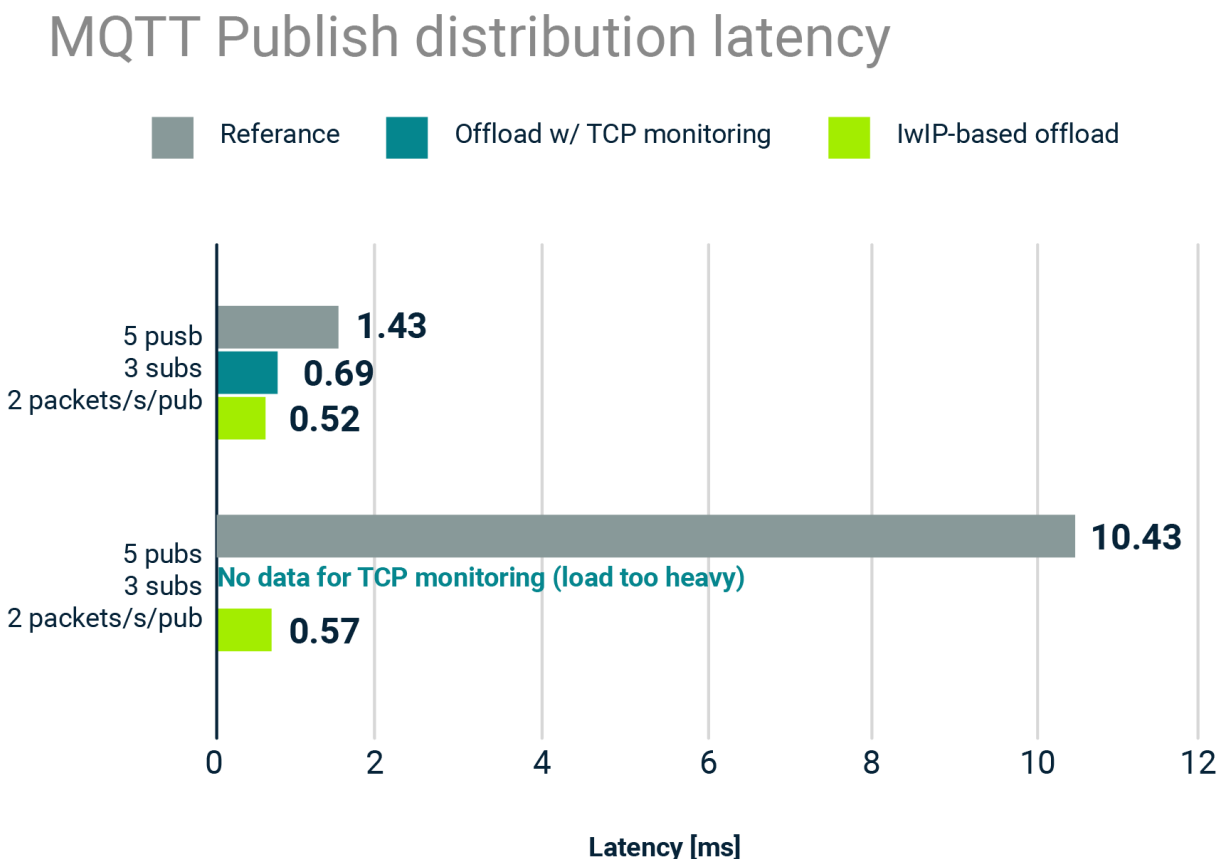
Problems addressed

As expected, incorporating a lwIP network stack into our MQTT protocol offload addresses all of the previously identified issues. We ought to be able to run tests exerting a heavier load and see how the protocol offload now fares.

Problem	Solved?	Details
SEQ/ACK desynchronization	Yes	Regular SEQ/ACK synchronization on separate connections (broker-offload, client-offload)
ACK matching	Yes	Problem does not exist with connection pairs
Simultaneous transmit	Yes	Tx always comes from the offload
Retransmissions	Yes	lwIP stack handles retransmissions
Delayed ACK	Yes	lwIP stack handles ACK processing
Nagle's algorithm	Yes	OlwIP stack buffers TCP data until a contiguous block is ready. Offload processes a contiguous data stream, message by message
Keep-alive	Yes	lwIP stack supports TCP keepalive
Congestion control	Yes	lwIP stack supports at least RFC 2581

Results

The chart you can see below contains the results from running two test configurations. The first one allowed us to compare how TCP monitoring and lwIP-based offloads performed, with the latter as the winner, which is surprising as the former is less complex, code-wise. The second one involved a larger number of publishers and subscribers to measure how good (or bad) the lwIP-based offload scales in contrast to a direct broker-client scenario. What was surprising is that the reference solution scaled poorly as the number of clients increased. However, we did not invest too much time in tweaking its default configuration.



Lessons learned & next steps

Overall, the performance of our solution seemed satisfactory. We have managed to significantly reduce the MQTT Publish packet distribution times. At this point, we could have concluded the proof of concept. However, we had noticed one problem that we want to tackle.

lwIP-based protocol offload introduces an additional network stack that packets traveling between the broker and the clients have to pass, which results in a latency increase on that path. For our MQTT protocol offload this problem is negligible since most of the MQTT traffic is related to Publish packet reception and distribution and occurs between the clients and the protocol offload. However, it may become significant for other TCP-based protocols where more packets have to be handled by the server (slow path). Hence, we have asked ourselves whether we can improve our design to address this problem.

Socket interception

In this iteration, we have focused on ways to bypass the kernel network stack on the host. The high-level goal seems similar to the previously mentioned ToE, but in our case, we still are going to accelerate the MQTT protocol.

Design

In order to eliminate the seemingly unnecessary host network stack from the packet path, we need to focus on exposing the network stack running on the DPU to the broker application. Essentially, in this approach we are facing two main problems: capturing relevant API calls issued by the broker and passing them to the DPU for processing.

Providing a socket interface

First, we need to provide the MQTT broker with a socket-like interface that will allow it to connect with clients. There are two approaches we can take:

Creating a libc shim

A libc shim means that we need to create a library that will provide the application with the Berkeley sockets API as it is implemented in the libc library present in the system. The calls to the sockets API need to be handled by the library. The sockets API is a frontend for many Linux subsystems and protocols; we not only have TCP sockets that will be handled internally, but Unix domain sockets or netlink, and most of those calls need to be handled by the Linux kernel.

Providing a custom library

Creating a custom library and using it in the broker directly is another way we could tackle the problem. That would of course require access to the source code of the broker, and at the very least rebuilding the application provided the sockets library would still replicate the libc API. Since this approach allows for more fine-grained control of the socket usage on the application level it is popular on the market; notable examples that use this method with great results are AMD's TCPDirect and F-Stack.

Forwarding calls to a DPU

Similarly there are many ways one could forward calls from the shim library to the DOCA application on a remote host. DOCA provides at least two libraries that we can use: DMA library and Comm Channel.

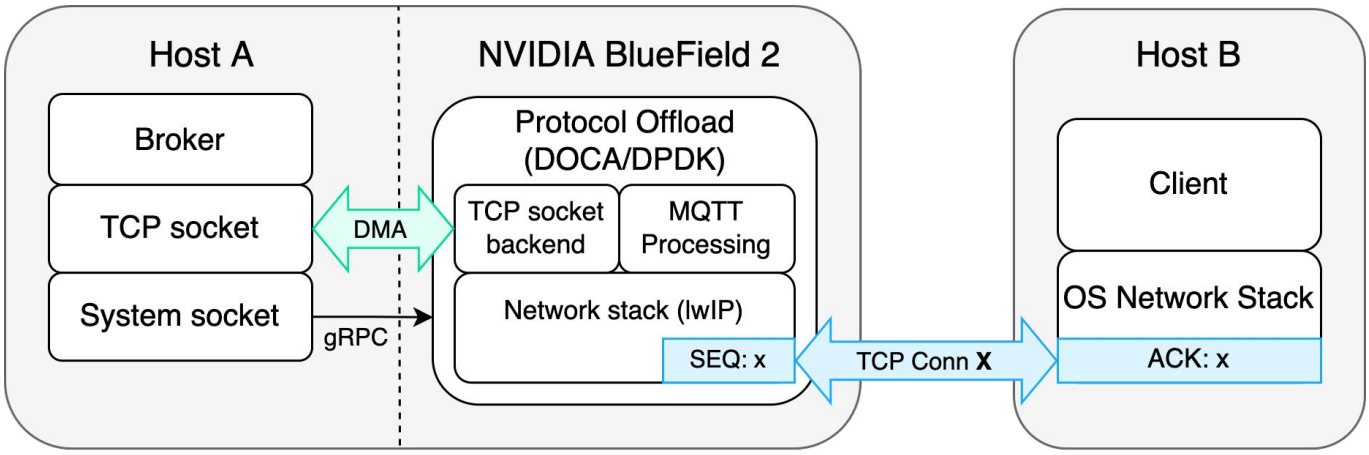
DOCA DMA library

This library provides essential components to access host memory from the DPU and vice versa. In theory, DMA would allow the DPU to access parameters passed from broker to socket API functions (i.e. Rx/Tx buffers). However, taking a deeper look into how those operations are performed for every memory buffer, the address, length and DOCA specific export needs to be sent from the host to the DPU. Consequently, DMA cannot be the only medium used to exchange data and we need a different method to send the DMA configuration. Furthermore, for small buffers, setting up DMA would be more costly than actually copying the data.

DOCA Comm Channel

Sending dynamically sized messages with textual content is used commonly enough that a library providing this functionality can be found among DOCA as well. The concept is rather simple - an application running on the DPU sets up a server side, while (multiple) applications on the host can connect as clients and interchange the messages.

Implementation



For the actual implementation, we have created a libc shim library that forwards interesting calls to protocol offload via the DOCA Comm Channel. The library running on the host is created in Rust and keeps the Comm Channel alive regardless of any traffic generated by the broker. Interestingly, DOCA Comm Channel initialization gathers information about available network devices by using netlink protocol and libc sockets. Since call interception is done for the whole process (which includes the DOCA Comm Channel library) we need to set up call forwarding to the native libc before we can initialize the Comm Channel.

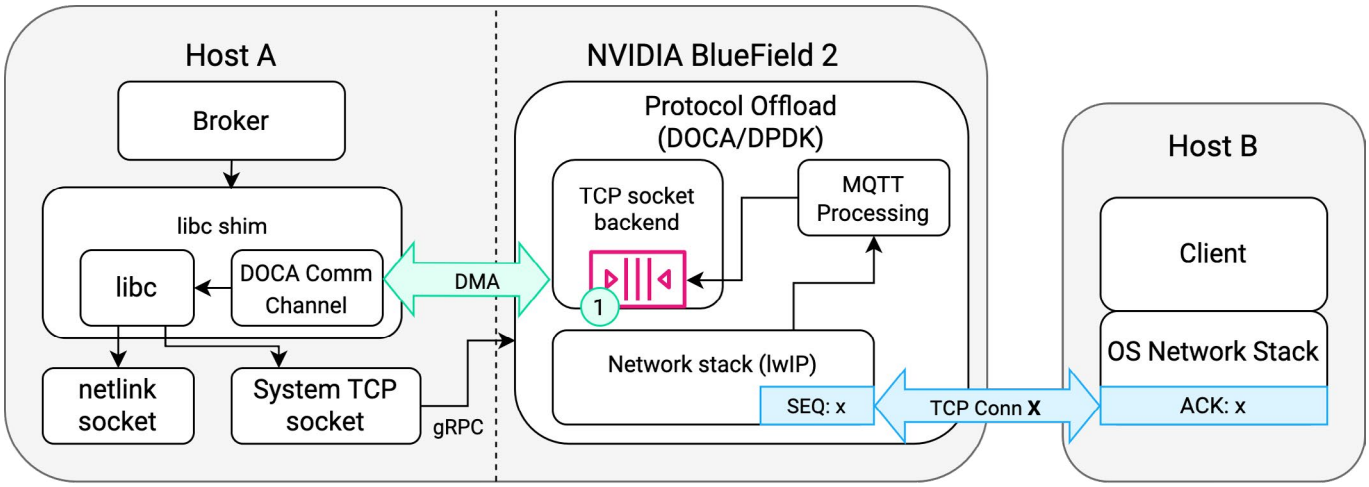
However, the Comm Channel is only a medium, and we need to specify the format for the messages we are going to exchange between libc shim and protocol offload, and for that, we have decided to use Cap'n Proto. This format is supported by libraries for both Rust and C++ and allows for easy integration.

Encountered issues

API compatibility

One of the issues worth mentioning is API compatibility between the DOCA network stack (in our case lwIP) and libc on the host. The application that uses sockets was written with libc in mind and expects compliance with libc documentation; any divergence can produce unexpected results. One of the integration steps in our projects was to run a simple ping from ip-utils and forward the calls to the DPU. Since both the host and DPU network stack ran the same version of Linux OS, we have not encountered any problems with this approach. However, once we moved to lwIP, we immediately crashed the ping since the implementation of ICMP in lwIP slightly differs from libc.

Packet cache



The next issue we needed to consider is that the TCP stream consists of MQTT messages that need to be handled by either protocol offload or by the broker on the host. While lwIP handles reassembly and defragmentation on the IP and TCP layer we still need to decide on how to deal with the actual MQTT protocol. In order to keep latency under control, the MQTT Processing block continuously polls on the lwIP socket internally representing the broker. Furthermore, the protocol offload implements a queue (1) related to that socket where it stores packets meant to be processed by the broker. On all recv() calls coming from the host, it provides bytes extracted from that queue.

Summary

We have made a case that implementing offloads for TCP-based protocols like MQTT requires that the guarantees provided by lower network layers (Ethernet, IP, TCP) also need to be satisfied. By outlining the various aspects that the protocol offload will need to take care of, we have shown that offloading TCP-based protocols is not an easy subject and that a lot of work is happening under the hood of contemporary TCP/IP network stacks.

This publication outlines a couple of ways a protocol offload can be implemented: either as a man-in-the-middle (MITM) operating on existing connections or as a separate TCP endpoint establishing connections on its own. It has also explained how the offload can be integrated with the host application and that it can be done completely transparently, without modifying the broker source code.

We have also demonstrated that our proofs of concept were successful under the assumptions we have posed on them. MQTT Publish packet distribution time was significantly reduced, as well as the broker host's CPU load.

We hope that this publication will facilitate further conversations and efforts related to realizing TCP-based protocol offloads on SmartNICs/DPUs/IPUs, possibly leading to production-grade solutions.

About the authors

Maciej Rabęda – Senior Software Engineer

Maciej is a senior software engineer at CodiLime with a decade-long journey in platform firmware and pre-OS networking.

His expertise lies at the intersection of hardware and software, focusing on DPDK and SmartNICs. In his recent projects, Maciej has optimized and enhanced network performance, contributing significantly to the evolution of next-generation networking solutions.

Beyond coding and networking, Maciej is a multifaceted individual with a diverse set of interests. A true tech enthusiast, he channels his passion into DIY electronics, where he explores the world of circuitry and hardware tinkering.

Outside of the tech sphere, he is a powerlifter with an additional focus on the sport of shooting.

Jan Zieleźnicki – Senior Software Engineer

Jan's primary focus revolves around the intricate world of Open vSwitch, where he orchestrates and refines network virtualization to meet the demands of modern computing. His proficiency extends to various DPDK applications, showcasing his versatility in harnessing high-performance data plane processing.

Jan, as a true enthusiast of Linux virtualization, thrives on exploring this open-source ecosystem's limitless possibilities, ensuring optimal functionality and performance in complex computing environments.

He finds relief and excitement in biking and hiking through the mountains in his spare time. Whether conquering challenging trails or cruising scenic routes, Jan's love for the outdoors is a testament to his adventurous spirit and commitment to a well-rounded lifestyle.

About CodiLime

Since 2011, CodiLime has been the engineering partner of choice for semiconductor companies, networking vendors, telecom services, and software solution providers.

We're home to 300 top-notch software developers, network engineers, DevOps experts, and solution architects. We appreciate long-term collaborations, above all, as well as our partners. Below are some names that have already trusted us:

CodiLime aims to link network engineering talent with business domain expertise – we focus on five N.E.E.D.S. - Networks, Equipment, Environment, Data and Security.

contact@codilime.com