An FPGA-based RoCEv2-RDMA Engine with DCQCN Congestion Control for the LST Advanced Camera Readout

IEEE Real Time Conference 2026

Filippo Marini, on behalf of the INFN Padova electronics design service

May 23, 2026

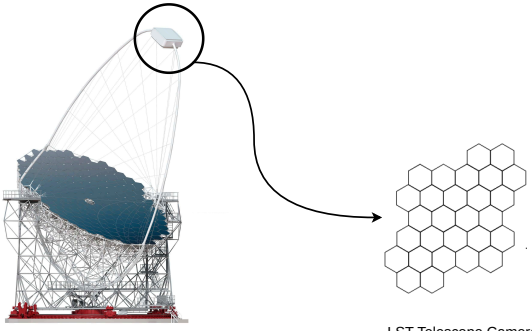
filippo.marini@pd.infn.it

INFN
Istituto Nazionale di Fisica Nucleare

Introduction

The LST Advanced Camera

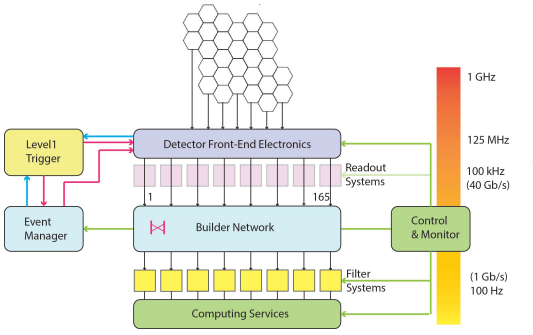
- The Large-Sized Telescopes (LSTs) cover the lowest-energy band of the observatory
- **Advanced Camera (AdvCam)**: upgrade based on **Silicon Photo Multipliers (SiPMs)**
  - $\sim$ 7000 pixels per camera ($\times 4$ vs. legacy PMT design)
  - Higher night sky background rate (in the order of GHz — stray photons from the sky picked up by the more sensitive SiPMs)
  - $\sim 10\times$ more data throughput
- Requires a fully **digital readout** with real-time hardware trigger



LST Telescope Camera

The Readout Architecture

- **Typical** HEP/astro DAQ chain: Front-End → **Readout Systems** → Builder Network → Filter / Computing
  - "Readout Systems" = a layer of **custom backend boards**: large FPGAs, many high-speed links, expensive
- **Our approach**: bypass that entire layer
  - The Front-End board does its own packetization, trigger, and **RDMA WRITE**
  - Goes **directly** into a commercial Ethernet fabric (the Builder Network)
- AdvCam Front-End: **49 SiPM channels** per board, 12-bit @ 1 Gsps, JESD204C at 12 Gb/s per lane, **10 GbE** output via RDMA



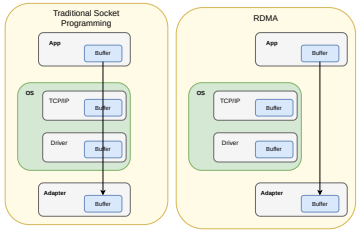
The "Readout Systems" layer is collapsed into the Front-End FPGA in our architecture

RDMA and RoCEv2

RDMA? RoCEv2?

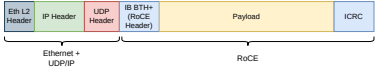
RDMA — Remote Direct Memory Access

- Direct **memory-to-memory** data transfer between hosts, **no CPU** involvement on either side
- Three founding concepts: **transport offload** (protocol stack in the NIC, not the OS), **kernel bypass** (user-space talks directly to the hardware), **zero-copy** (straight from app memory to the wire)
- **Low latency**, **very high throughput**, **near-zero CPU load**



RoCEv2 — RDMA over Converged Ethernet v2

- Specified by the InfiniBand Trade Association (IBTA)
- RDMA semantics on top of standard **UDP/IP/Ethernet** → routable, runs on **COTS** switches and NICs

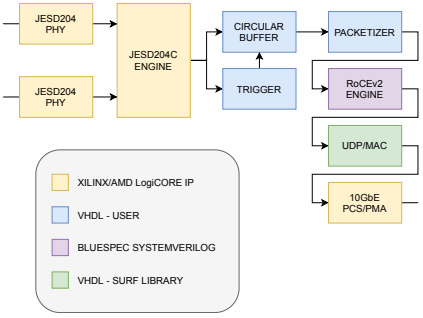


RoCEv2 reliability

UDP is connectionless — RoCEv2 layers a **reliability protocol** (PSN, ACK/NAK, retransmission, in-order delivery) on top, used in **Reliable Connection (RC)** mode

RoCEv2 Core in FPGA

- Based on an **open-source RoCEv2 core** [DatenLord, blue-rdma]
  - Designed to perform as Network Interface in a server, targeting very large FPGAs with massive external RAM
- Written in **Bluespec SystemVerilog (BSV)**
  - Higher-level than VHDL/Verilog, more hardware control than HLS
  - bsc compiler (open-source) emits plain Verilog — vendor-agnostic
- We **customized** the core for our needs:
  - Targeted at small FPGAs with no external BRAM
  - Removed entire **RDMA receive datapath**
- Added a **congestion management** module
  - Written in **VHDL** for better integration with the SURF network stack (rate limiter + DCQCN reaction)
- Customized the network stack from **SLAC SURF** (UDP/MAC/10GbE):
  - Insert/validate the **iCRC** on the data path
  - Thanks to Larry Ruckman and the SURF team for their support!



Resource (XCKU040)	Original	Modified
LUT	92475 (38%)	29802 (12%)
FF	136599 (28%)	40902 (8%)
BRAM	18 (3%)	8.5 (1%)

$\sim 3\times$ less LUTs/FFs, $\sim 2\times$ less BRAM on XCKU040

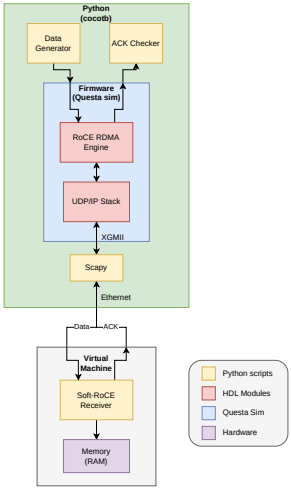
Firmware Simulation

RoCEv2 Firmware Simulation Setup

- Goal: **end-to-end verification** of the RoCEv2 engine + UDP/MAC stack, before going to hardware
- Stack of tools:
  - **Questa Advanced Simulator** runs the HDL
  - **cocotb**: Python coroutine library that drives the testbench
  - **cocotbext-eth**: parses the XGMII output of the SURF MAC into full Ethernet frames
  - **Scapy**: injects/receives the frames on a real network interface
  - **Target**: a Virtual Machine running **Soft-RoCE** (kernel software implementation of RDMA)
  - **Pyverbs** script on the VM acts as the **RDMA receiver**
- Connection mode: **Reliable Connection (RC)** → ACKs flow back

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.56.12	192.168.56.100	RRoCE	110	RC RDMA Write Only QP=0x000173
2	0.000333	192.168.56.100	192.168.56.12	RRoCE	62	RC Acknowledge QP=0x000011

Simulation packets in Wireshark



Performance Tests

Hardware Performance Tests

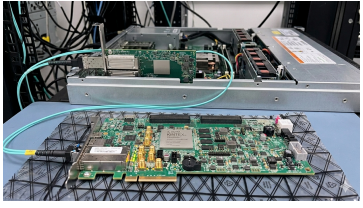
- Firmware deployed on **Xilinx/AMD KCU105**
- Throughput measurement and data integrity check

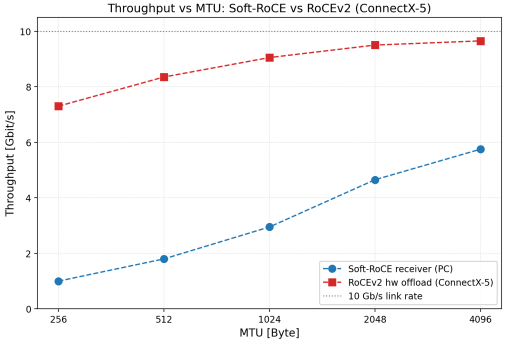
Test 1 — Custom core to Soft-RoCE receiver

- Bottlenecked by the **receiver CPU**: ~ 5 Gb/s on a 10 Gb/s link

Test 2 — Custom core to server with Mellanox ConnectX-5 NIC

- Hardware RDMA offload on the receiver: $\sim$ **9.7 Gb/s** sustained, essentially **line rate**





Soft-RoCE and RoCEv2 throughput comparison

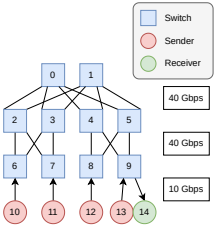
- Throughput grows with MTU (header overhead becomes negligible)
- Best operating point: **MTU = 4096 B**

DCQCN Congestion Control

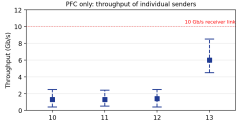
Why Congestion Control? — PFC is Not Fair

- The AdvCam DAQ is a **funnel** (incast): many 10 GbE BE links → fewer switch uplinks → servers
- RDMA assumes a **lossless** fabric — even rare drops cause RoCE retransmissions and **throughput collapse**

- The classical lossless mechanism is **PFC** (Priority Flow Control): when a switch port fills up, it sends **PAUSE frames** upstream
- PFC stops losses, but it has well-known problems:
  - **Head-of-line blocking**: innocent flows get punished too
  - **Unfair**: senders gets paused per link, not per flow



Typical data center network topology



PFC unfairness

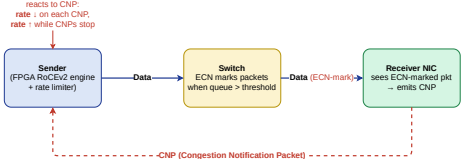
- DCQCN allows for a **finer-grained, end-to-end, per-flow reaction**

DCQCN — End-to-End, Per-Flow Reaction

- **DCQCN** = Data Center Quantized Congestion Notification [Zhu et al., SIGCOMM 2015]
- Standard congestion-control algorithm in commercial RDMA deployments
- **Complementary to PFC**: fabric lossless *and* fair

How it works

1. Switch **marks packets with ECN** when its queue exceeds a threshold (no drops, no pauses)
2. The **receiver** sees ECN-marked packets → sends a **CNP** back to the offending sender
3. The **sender** reacts: sharp decrease on each CNP, gradual recovery while CNPs stop arriving



Rate-control equations

Congestion estimator α (updated per CNP)

$$\alpha \leftarrow (1-g)\,\alpha + g, \qquad 0 < g \ll 1$$

with decay toward 0 if no CNP arrives within τ_0:

$$\alpha \leftarrow (1-g)\,\alpha$$

Current Rate (R_C) and Target Rate (R_T) on CNP arrival

$$R_T \leftarrow R_C, \qquad R_C \leftarrow R_C\,(1-\tfrac{\alpha}{2})$$

Rate recovery — three phases, gated by a timer + byte counter against threshold F

- *Fast Recovery* (counters $< F$): $R_C \leftarrow (R_C + R_T)/2$
- *Additive Increase* (counters $\geq F$): $R_T \leftarrow R_T + R_{AI}$, $R_C \leftarrow (R_C + R_T)/2$
- *Hyper-Additive Increase*: $R_T \leftarrow R_T + R_{HAI}$, $R_C \leftarrow (R_C + R_T)/2$

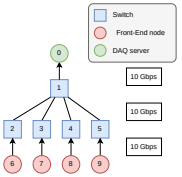
- Per-flow reaction → **only the right senders slow down**
- Net effect: **lossless and fair** share at the bottleneck

RoCEv2 + DCQCN in ns-3

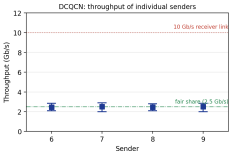
- **Before** developing any firmware, we built a full **ns-3 simulation** of RoCEv2 + DCQCN
- Topology modeled after a realistic (yet simplified) AdvCam-like fabric: multiple senders converging onto one receiver

What ns-3 gives us

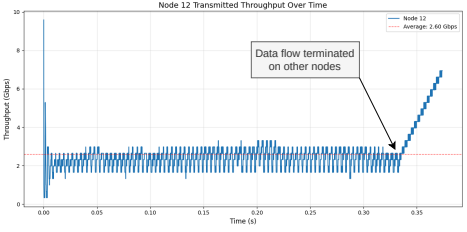
- Validates the **DCQCN dynamics** at the protocol level, independent of hardware
- Lets us **explore the parameter space** (ECN thresholds K_{min}/K_{max}, rate-decrease factor, rate-increase timer, RAI, F, etc.)
- Identifies the optimal **operating point** for the real-world experiment conditions



Simulated network topology



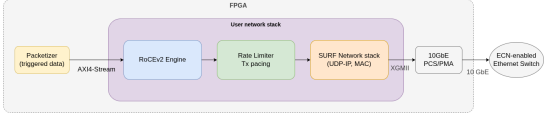
DCQCN fairness



ns-3 throughput vs time

DCQCN in Our FPGA Stack

- SURF MAC: process inbound CNPs, detect ECN-marked ACKs
- **Rate limiter** with the DCQCN rate-decrease / rate-increase state machine
- **Parameters** can be set **run-time**
- Target: every front-end board behaves like a well-behaved tenant on an ECN-enabled fabric, like any commercial RDMA NIC



Full network stack in FPGA

Resource	Used	Percentage
LUT	1337	0.5%
FF	2557	0.5%
BRAM	18	3.0%
DSP	5	0.3%

DCQCN module footprint (XCKU040)

- **Negligible** area overhead on top of the customized RoCEv2 core

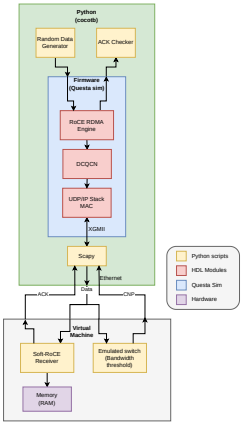
DCQCN Firmware Simulation

- Same backbone as the RoCEv2 simulation: **cocotb + Scapy + Questa**, target = **VM with Soft-RoCE** receiver
- New ingredient on the receiver side: an **"emulated switch" written in Python**

The emulated switch does two things:

1. Measures the **throughput** of the incoming RoCEv2 stream
2. When throughput exceeds a configurable **threshold**, it **forges CNP packets** and sends them back to the simulated FPGA

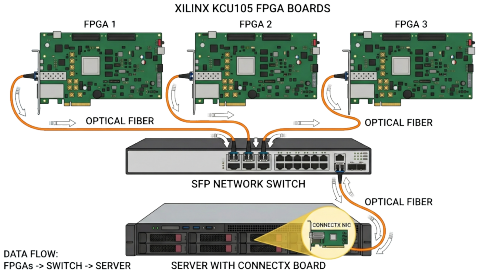
DCQCN simulation dashboard. Actual throughput (blue line) and fake switch threshold (orange dotted line) over time. CNP packets received are marked by the vertical green/yellow dotted line



DCQCN Hardware Test

Hardware Test Setup

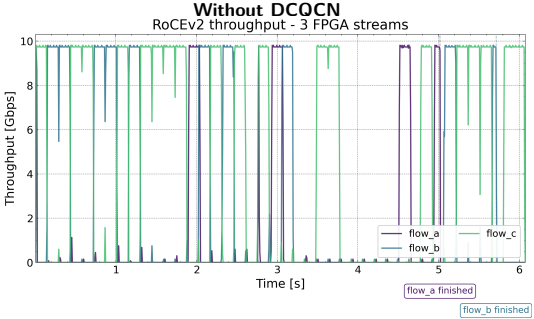
- **Three KCU105 boards** acting as **RoCEv2 senders**, each on a 10 GbE link
- All three feed a **DCQCN-enabled Ethernet switch**
- Switch uplink goes to a **server equipped with a Mellanox ConnectX NIC** acting as the **single receiver**, also on a 10 GbE port



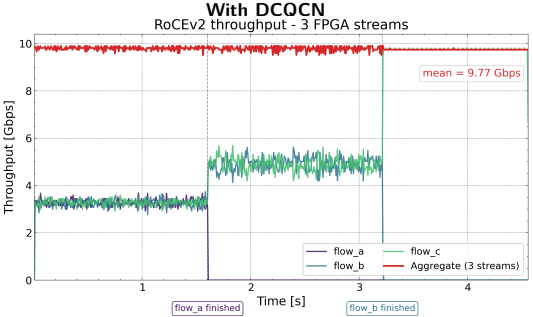
- Switch is configured with **ECN marking thresholds** on the egress queue toward the receiver
- No firmware changes between senders: same modified RoCEv2 + DCQCN-aware MAC on all three

- Classic **incast** scenario: 3×10 Gb/s offered, 1×10 Gb/s sink

DCQCN vs. no DCQCN — Hardware Results



Chaotic on/off pattern: each sender alternately monopolizes the link while the others are stalled by PFC + RoCE retransmissions.

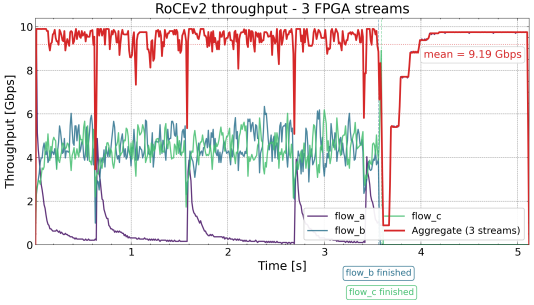


Smooth fair share: 3 → 2 → 1 senders, identical DCQCN parameters, no losses, reactive redistribution.

- Same hardware setup, same RoCEv2 traffic, **only DCQCN toggled**
- Without DCQCN: senders take turns being starved → unfair, bursty, retransmission-heavy
- With DCQCN: per-flow ECN → CNP feedback → **fair sharing + reactive redistribution, lossless**

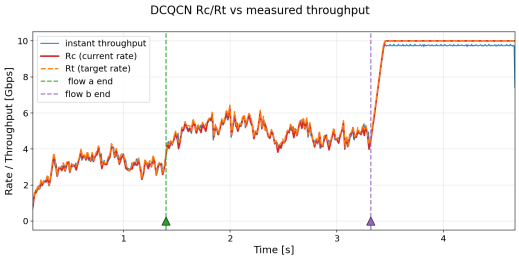
⇒ **Our FPGA RoCEv2 + DCQCN reproduces commercial-NIC behavior on a real ECN-enabled fabric**

Rate Limiter Verification



Algorithm behavior (qualitative)

- Three senders with **different DCQCN parameters** — one with deliberately long intervals
- The state machine becomes **visible**: the textbook **saw-tooth**
- Confirms the FPGA reproduces DCQCN's expected *qualitative* dynamics



Rate limiter ↔ actual throughput (quantitative)

- Live monitor of the **internal** R_C **register** of the DCQCN rate limiter, plotted alongside the **measured RoCEv2 throughput**
- The two curves **overlap throughout the run**
- The FPGA's actual link rate **tracks** R_C cycle-by-cycle — the rate limiter is the *sole* governor of throughput, no hidden bottlenecks

Conclusions

Summary and Outlook

What was achieved

- Customized, lightweight **RoCEv2 RDMA core** in BSV, integrated with the SLAC SURF UDP/MAC stack
- Full **cocotb + Scapy + Soft-RoCE** simulation flow, validating both protocol and datapath end-to-end
- **9.7 Gb/s** line-rate RDMA WRITE from KCU105 to a ConnectX-5 server
- **DCQCN** congestion control developed on top:
  - Verified in simulation against a Python fake switch
  - Proven in hardware on a **3 senders → 1 receiver** incast over a real DCQCN-enabled switch
  - Correct **algorithm dynamics**, **fair bandwidth sharing** and **reactive redistribution** as senders join/leave

Outlook

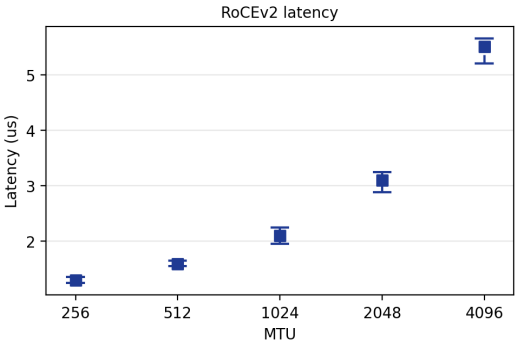
- Integrate the DCQCN-enhanced core into the full AdvCam readout chain
  - Direct path SiPMs → off-the-shelf ECN-capable switch → server
- Open-source contributions back to SURF (iCRC already upstream; DCQCN to follow)

Thank you!

filippo.marini@pd.infn.it

Backup

RoCEv2 Round-Trip Latency



Latency vs MTU

- **Latency definition**: full round trip as seen by the FPGA application — from posting a Work Request (WR) to receiving its Work Completion (WC), **including the DMA fetch of the payload from FPGA BRAM** and the ACK return
- Measured on the **FPGA** $\leftrightarrow$ **ConnectX-5** hardware setup
- **AdvCam readout is throughput-bound, not latency-bound** $\rightarrow$ we deliberately choose **MTU = 4096 B** to maximize throughput; the $\sim 4\times$ latency penalty over MTU $=$ 256 B is acceptable for our use case