Design of Streaming Protocol for Real-Time Reliable RFSoC Streaming

KISHORE RAJENDRAN, RUSS SOBTI, and DANIEL ZIPER, UC San Diego, USA

Current and future wireless communication system architecture demands extremely high data throughput, low latency, and reliable real-time connectivity. Traditional Software Defined Radio (SDR) platforms, such as the USRP, struggle to meet the needs of 5GNR and beyond's needs due to their bandwidth limitations, processing bottlenecks, and software limitations. AMD/Xilinx's RFSoC line is a promising platform for developing modern communication systems, with its performant ADC/DAC modules, fast FPGA, and soft-core processor support in one board. However, the base RFSoC ecosystem lacks a streaming protocol capable of handling bidirectional, high-bitrate data streaming between the unit and its host machine.

This paper addresses these limitations by developing a custom, high-throughput, UDP-based streaming protocol [1] utilizing direct SFP+ Ethernet connections. The proposed solution leverages a lightweight, FPGA-friendly "Pseudo-TCP" approach employing Negative Acknowledgments (NACKs) and sliding window techniques to achieve reliable, low-latency data transmission suitable for RF data streaming. Our implementation emphasizes modularity, portability, and efficient resource utilization on FPGA platforms by avoiding dynamic memory allocation and employing static polymorphism through templated C++ components.

Experimental evaluations demonstrate the protocol's robustness and performance via extensive loopback testing, achieving substantial throughput improvements over traditional SDR approaches. The initial results of the implementation and benchmarking indicate successful protocol operations with further potential for optimization. By bridging the gap between RFSoC hardware capabilities and practical data streaming requirements, this work enables researchers and engineers to realize advanced, scalable, and real-time capable systems crucial for next-generation radio access network (RAN) research and deployment.

Additional Key Words and Phrases: RFSoC, FPGA, SDR, High-Speed Data Streaming, UDP, TCP, NACK, Sliding Window, Ethernet, Real-Time, Streaming, 5G NR, AXI Stream

ACM Reference Format:
Kishore Rajendran, Russ Sobti, and Daniel Ziper. 2025. Design of Streaming Protocol for Real-Time Reliable RFSoC Streaming. 1, 1 (March 2025), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Motivation

Modern wireless communication systems, particularly among 5G New Radio (5G NR) and emerging 6G networks, require unprecedented levels of data throughput, minimal latency, and robust real-time connectivity to handle increasingly complex applications such as Massive MIMO, mmWave communication, and advanced mobility management. Current commercial Software-Defined Radio (SDR) platforms, such as USRP, exhibit significant limitations in capability to handle bandwidth and reliably provide I/Q data from the appropriate constellations in real-time. Notably, USRP leverages raw UDP to transmit I/Q data, which leads to packet loss when attempting to transmit the data required for modern workflows.

1.2 Problem Statement

The AMD/Xilinx RF System on Chip, or RFSoC, has the potential to address many of the shortcomings of the USRP through exposing high-speed ADCs and DACs, powerful FPGA capability, and an embedded soft-core processor. However, the RFSoC is an FPGA-driven solution and does not come with a well-defined protocol or toolchain for onboarding and offloading data in real-time, hampering its utility beyond testing with pre-defined packets. The core challenge is to develop a reliable, real-time streaming protocol capable of handling bidirectional, high-bandwidth RF data streaming between the RFSoC and its host computer.

1.3 Prior Work

Previous attempts towards creating reliable streaming protocols, such as webRTC, and RDMA-based solutions, such as RoCE and iWARP, have either been overly complex or unsuitable for FPGA environments due to dependencies on extensive software stacks and kernel overhead. While there have been promising developments in UDP-based approaches for real-time embedded streaming, existing solutions remain limited either by scalability issues, unidirectional constraints, or their specificity to non-RF applications.

1.4 Key Ideas

To address these gaps, this paper proposes a novel UDP-based streaming protocol tailored specifically to RFSoC and FPGA environments. Our protocol integrates core concepts of TCP to achieve reliability, while significantly reducing protocol overhead by designing a custom sliding window mechanism with Negative Acknowledgment support.

1.5 Summary of Results and Contributions

Our experimental evaluations, conducted through comprehensive loopback testing, demonstrate that our custom streaming protocol achieves substantial throughput improvements and superior reliability compared to conventional SDR streaming techniques. We provide a detailed description of the protocol implementation, hardware and software evaluation methodologies, and practical insights into addressing real-time constraints on FPGA platforms. By significantly enhancing the streaming capability of a base RFSoC, this work contributes a robust solution for ongoing research into next-gen wireless communication systems and radio access networks.

Authors' Contact Information: Kishore Rajendran, kirajendran@ucsd.edu; Russ Sobti, rsobti@ucsd.edu; Daniel Ziper, dziper@ucsd.edu, UC San Diego, San Diego, California, USA.

ACM XXXX-XXXX/2025/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Background and Related Work

In this section, we review key literature and industry practices in streaming protocol design, examine existing protocols used for streaming data, and discuss the emergence of the RFSoC as a versatile SDR platform.

2.1 Streaming Protocol Design

Streaming protocols must manage conflicting requirements to ensure high data throughput with low latency while ensuring reliability. Streaming protocols are often designed around a core application type, striking a balance between latency, throughput, reliability, and jitter. In theory, ensuring the delivery of every packet in-order is possible, but this incurs significant delay and overhead. However, certain applications might prioritize speed versus reliability, and thus simply send data without verifying its receipt on the other end. Noronha et al. discuss these tradeoffs by demonstrating how protocols such as TCP, which enforce in-order and reliable delivery, introduce significant delays due to head-of-line blocking and congestion control mechanisms.[8] Conversely UDP's minimal overhead is much faster, however it may be lossy. TCP and UDP are the two extreme implementations - TCP ensures scalability across the Internet scale through congestion control and reliability through its sliding window, but at the cost of speed. UDP ensures speed and is minimal, however it does not make any guarantees about coherence of data.

Protocols like QUIC and RDMA-based approaches attempt to bridge this gap. QUIC, for example, maintain many of the reliability features of TCP while operating over UDP, thus reducing connection setup and retransmission delays. [6] RDMA-based solutions often bypass the OS's networking stack altogether, achieving microsecond latencies by directly transferring data between memory-mapped regions[10]. However, these techniques require custom drivers and/or supported hardware, which makes these implementations unsuitable for some applications. Hybrid approaches, such as Secure Reliable Transport (SRT) add selective retransmission and forward error correction atop UDP, which minimizes the latency impact associated with TCP while still providing reliability. [5] Additionally, for reliable streams going to more than one recipient, protocols like the Naval Research Laboratory's NACK-Oriented Multicast (NORM) offer low-latency reliable multicast streams over a UDP base. In summary, there are many streaming protocols in existence, and the optimal design or selection of streaming protocol depends on the application's tolerance for loss and delay, the expected number of concurrent streams, and the application's expected throughput. There is not necessarily a "one-size-fits-all" design that is optimal for all streaming use cases.

2.2 Streaming Protocols for RF data

The practical implementation of streaming in the Software-Defined Radio domain further illustrates these tradeoffs. Ettus Research's USRP devices, for example, employ their USRP Hardware Driver (UHD) framework which leverages the VITA49.2 standard for RF data encapsulation and UDP for transport of I/Q data over the wire. UHD organizes raw IQ samples into packets that include essential metadata, including sequence numbers, timestamps, and framing information, supporting real-time processing.[9] The use of UDP for transport minimizes overhead per packet, allowing the system to handle high sample rates, albeit with packet loss. When a packet is dropped, the sequence numbers let the application mark losses, however UHD does not support retransmission.

UHD's implementation mimics the design decisions made when designing a streaming protocol for the Audio-Visual (A/V) domain. In the A/V domain, protocols like RTP (Real Time Transport Protocol) similarly prioritize low latency over data integrity. Research into A/V streaming is highly applicable to SDR as well - parallels exist between the ever-increasing bitrates and resolutions of video data and the increasing desire for high-bandwidth sampling with software-defined radios. SDR and A/V protocols both have started to implement adaptive bitrate adjustments and selective retransmission, which are used to manage real-time needs while ensuring reliability. [7]

2.3 RFSoC as an SDR Platform

Xilinx's RFSoC platform is a significant evolution in the SDR ecosystem, supporting multi-gigabit ADCs/DACs, FPGA fabric, and embedded processors on a single chip. This integration simplifies the design of high-bandwidth and MIMO SDR systems by reducing the need for multiple discrete components. Xilinx's documentation shows the applications to 5G NR and Massive MIMO deployments, where the consolidation of RF Frontend components into one device not only minimizes hardware complexity, but also improves synchronization across channels.

Several studies have shown RFSoC to be effective in practical use cases. Redondo et al. [2] describe an RFSoC-based readout system for microwave telescopes, showing state-of-the-art noise performance while maintaining a wide bandwidth. Similarly, Gartmann et.al [3] compare RFSoC to traditional SDR setups for high-bandwidth applications, showing RFSoC to meet real-time processing demands while reducing overall latency by the elimination of serialization links.

3 Design, Testing, and Implementation

Our design focuses on implementing a robust, high-throughput streaming protocol that bridges the RFSoC hardware and the host PC for real-time RF data transmission. The overall approach is structured into a hardware workflow that tests the streaming protocol on the FPGA, a software streaming workflow that handles data transport and error recovery, and a protocol implementation defined via finite state machines (FSMs).

3.1 System Architecture and Initial Setup

The system is built around the Xilinx ZCU111 RFSoC, which interfaces with a host PC over a 10G/25G Ethernet link via SFP+ ports. To validate our approach, an initial test setup is implemented using a custom synthetic data generator instead of real IQ data from the RFDC block. This strategy simplifies debugging by allowing us to monitor counter values rather than complex RF signals. As compared to the end goal system design in Figure 1, for the initial setup we used an Integrated Logic Analyzer (ILA) to capture FPGA data while Wireshark monitors incoming packets on the host PC.

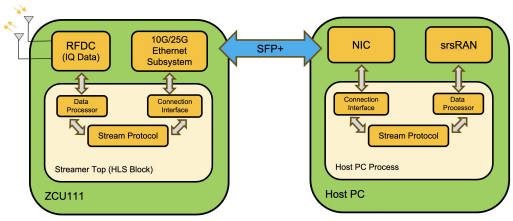

Fig. 1. System Block Diagram

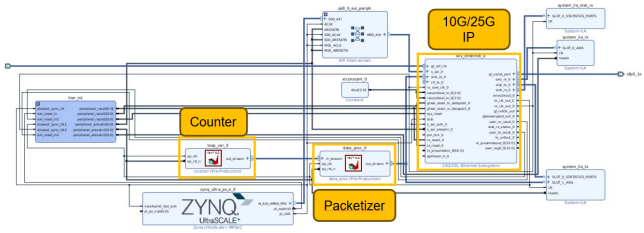

Fig. 3. Vivado Block Design for unidirectional SISO stream

3.2 Hardware Workflow: FPGA Implementation

The FPGA workflow is divided into three key parts:

3.2.1 Custom Synthetic Data Generator. A custom synthetic data generator, implemented as a simple up counter (0–255 cyclic), is developed using Vitis HLS. This block utilizes the AXI Stream protocol—the same protocol used by the 10G/25G Ethernet subsystem IP—to format data for transmission over the SFP+ ports. Figure 2 displays the counter output captured by the ILA on the ZCU111 board.

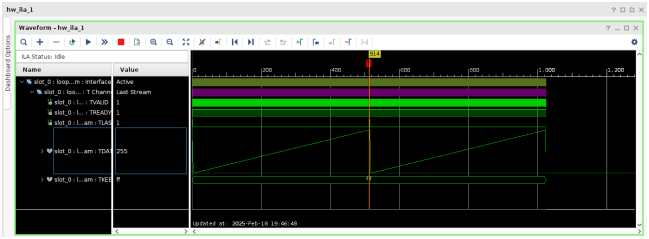

Fig. 2. HLS Counter running on ZCU111: Output on ILA

3.2.2 Custom Packetizer Block. A custom data packetizing block is implemented using Vitis HLS. This block takes the incoming AXI Stream data, breaks it down into packets and adds the required headers. Essentially, this block is responsible for taking the continuous stream of incoming data and breaking it down into packets along with adding appropriate headers, so that the Ethernet Subsystem is able to transmit these packets over the SFP+ ports.

3.2.3 10G/25G Ethernet Subsystem Integration. To interface with the four onboard SFP+ ports of the ZCU111, we integrate the 10G/25G Ethernet Subsystem IP [4] into our design. This IP core, sourced from AMD, facilitates the creation of block designs in Vivado that manage data transfer between the programmable logic (PL) of the FPGA and the physical SFP+ interfaces. As depicted in Figure 3, the current unidirectional SISO (Single Input Single Output) streaming setup routes counter data from the custom data generator block to the custom packetizer block and then finally through to the Ethernet IP for transmission. This data can then be analyzed on the host PC using tools like Wireshark.

As a simple test before we actually deploy our protocol, we send this synthetic data using simple headers (just involving source and destination addresses along with ethernet type) in broadcast mode. Figure 4 shows the results observed on the ILA waveforms monitoring the transmit data statistics from the Ethernet Subsystem IP. Upon calculating the frequency of good packets based on the clock frequency, we see that the transmission of good packets is around 2.44 Gbps from the ZCU111, which is quite impressive. This is further discussed in the evaluations.

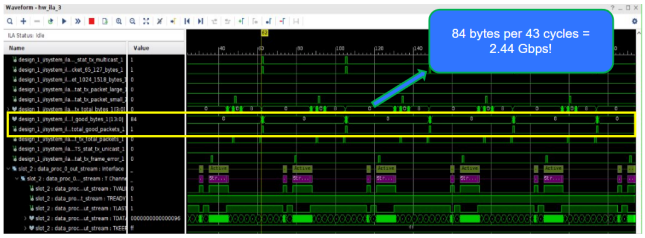

Fig. 4. ILA waveforms showing successful transmission of good packets

3.3 Software Streaming Workflow

The software side of the design implements our custom streaming protocol using a modular approach that mirrors the FPGA implementation. The protocol is built around a sliding window mechanism for reliability, where each packet carries a unique sequence number. The sender transmits packets continuously until the window limit is reached, and then waits for feedback from the receiver.

Figure 5 outlines the sequence of operations:

- The FPGA sender initiates data streaming, tagging each packet with a sequence number.
- The receiver processes incoming packets and sends back ACKs for the highest contiguous sequence received.
- In the event of a missing or out-of-order packet, the receiver issues a Negative Acknowledgement (NACK), prompting the sender to retransmit only the missing packet.
- If no acknowledgment is received within a set timeout, the sender retransmits all unacknowledged packets.
- A FIN/FIN-ACK exchange (not shown) gracefully terminates the session.

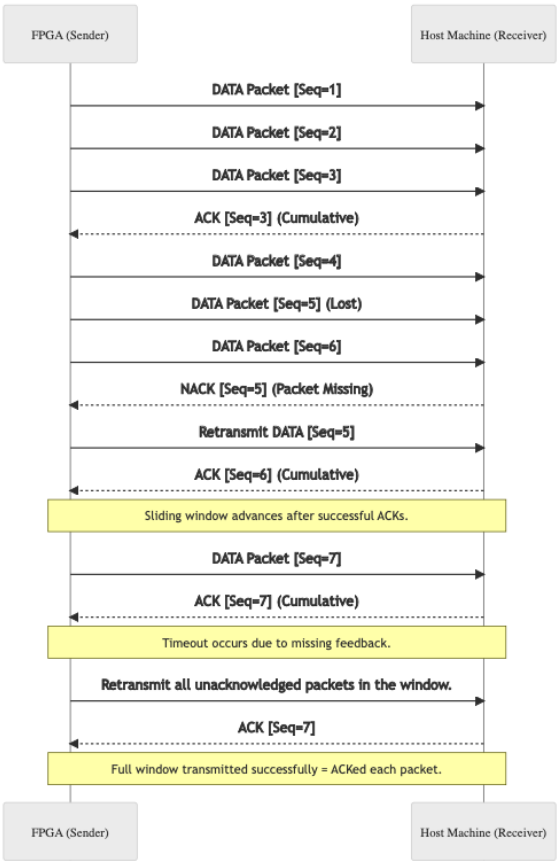

Fig. 5. Sequence Diagram for Reliable UDP Streaming

Furthermore, Figure 6 presents the loopback testing block diagram, which demonstrates how the various software components (Stream Protocol, Data Processor, and Connection Interface) interconnect in a host PC context. The Streaming Protocol implementation is designed to be fully compatible with HLS, with no memory allocation or polymorphism. C++ templates are used for static "polymorphism" to allow for the Data Processor and Connection Interface abstractions. The Data Processor abstracts Data I/O, such as reading/writing to a file, reading IQ samples on the FPGA, or interfacing with srsRAN through ZMQ. The Connection Interface abstracts sending/receiving data from the network, such as interfacing with the Ethernet Subsystem on the RFSoC, or opening a UDP socket on the PC. This design enables full code reuse of the Stream Protocol component across FPGA and host implementations and ensures that our protocol can be validated thoroughly in a loopback configuration before deployment in real-world RF streaming scenarios.

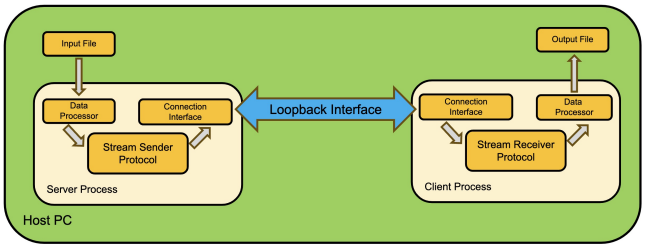

Fig. 6. Loopback Testing Block Diagram

3.4 Finite State Machine Models for Protocol Implementation

To manage the complexity of reliable streaming over an unreliable transport (UDP), our protocol logic is structured using finite state machines:

3.4.1 Sender State Machine. The Sender FSM begins in an idle state, awaiting a Start Stream Command from the receiver. Once initiated, it transitions to the Sending state, where it transmits packets sequentially, each with an assigned sequence number. Upon reaching the end of the sliding window, the sender enters the WaitingForACK state to monitor for acknowledgments. Depending on the feedback:

- A received ACK shifts the window forward.
- A NACK triggers retransmission of the missing packet(s).
- Absence of feedback beyond a timeout results in retransmission of all unacknowledged packets.

If multiple timeouts occur, the sender may revert to the Idle state to terminate the session.

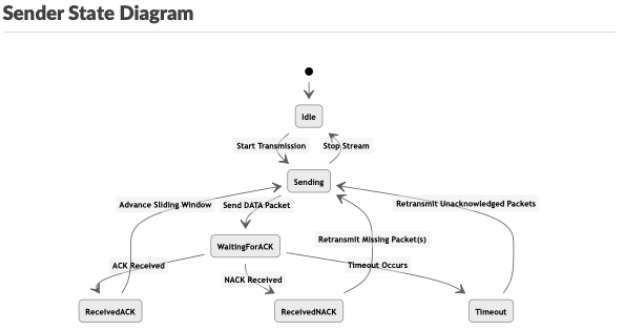

Fig. 7. Stream Sender State Diagram

3.4.2 Receiver State Machine. The Receiver FSM starts in the Idle state by sending a StartStream command. It then transitions to the Receiving state to process incoming packets. The receiver verifies packet sequence numbers and checks data integrity:

- Correctly ordered and verified packets move the receiver to the Processing state, where the packet is stored.
- Once the highest contiguous sequence is reached, an ACK is sent.

- If a packet is missing or arrives out-of-order, the receiver issues a NACK to request retransmission.

This loop continues until a FIN packet is received, prompting a return to the Idle state.

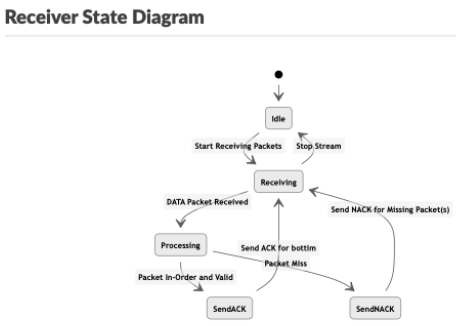

Fig. 8. Stream Receiver State Diagram

3.5 Summary of Design Approach

Our design leverages the integrated capabilities of the RFSoC platform by:

- Utilizing a synthetic data generator for early-stage debugging and validation.
- Implementing a modular FPGA workflow that includes dedicated blocks for data generation, Ethernet interfacing, and protocol management.
- Designing a flexible software streaming workflow that mirrors the hardware implementation to facilitate loopback testing and code reuse.
- Structuring the protocol using finite state machines to handle packet transmission, feedback processing, and error recovery in a reliable yet low-latency manner.

This holistic approach ensures that the system can achieve high-throughput, real-time RF streaming while providing the flexibility to scale and adapt to different data sources and network conditions.

4 Evaluation

4.1 Ethernet Subsystem Throughput Benchmarking

Initial testing was conducted using a synthetic data generator and simple packetizer implemented on the FPGA. The packetizer was connected to the Ethernet Subsystem, whose output was monitored by an ILA. As shown in 4, the Ethernet Subsystem reported it was sending data at almost 2.5 Gbps, however we were unable to receive this data on the PC.

4.2 Loopback Testing and Throughput Benchmarking

To validate the reliability of our software implementation, and benchmark the maximum throughput of our software stack, we ran the Streamer and Receiver processes over the loopback interface on the Host PC. By randomly inducing bit flips in some proportion of received packets, we can easily simulate low channel quality or congestion. When a bit is flipped in a received packet, the checksum validation in our protocol fails, and the packet is dropped, and a NACK will later be sent for this missing packet. By streaming a file across the interface with simulated errors and then checking the received file matches the input file, we verified our streaming protocol is 100% reliable like TCP. We tested a range of window sizes for different simulated error probabilities in order to determine the maximum throughput. Fig-9 shows how optimal window size for our protocol depends on the channel quality. For 0.0001 probability of packet drops, we were able to achieve 2689 Mbps effective throughput over loopback, with a window size of 800 packets.

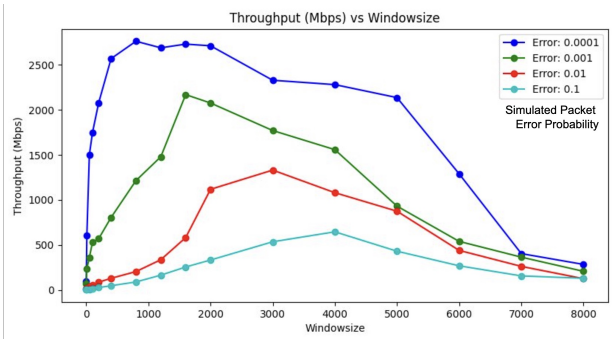

Fig. 9. Throughput of streaming over Loopback, with various simulated probability of packet error.

4.3 Real-World Testing over LAN and NIC Interfaces

In addition to the controlled loopback environment, further tests were performed over a Local Area Network (LAN) using two host PCs connected via a network interface. The setup is illustrated in Figure 10. These tests aimed to mimic a real-world scenario where the RFSoC and host are in separate physical locations. Although achieving consistent throughput proved challenging due to additional network variables, the tests provided valuable insights into packet loss behavior and network-induced latency. Our stream receiver, Wireshark and iperf3 were used to benchmark performance. We were able to achieve 955 Mbps throughput over the network, which is between the throughputs of TCP (940 Mbps) and UDP (959 Mbps) as tested with iperf3, while maintaining the 100% reliability of TCP.

We also compared our protocol to UDP over an 10 Gbps SFP+ connection between 2 PCs. We achieved 5.5 Gbps using our Streaming Protocol, compared to 6.5 Gbps with UDP, as tested with iperf3.

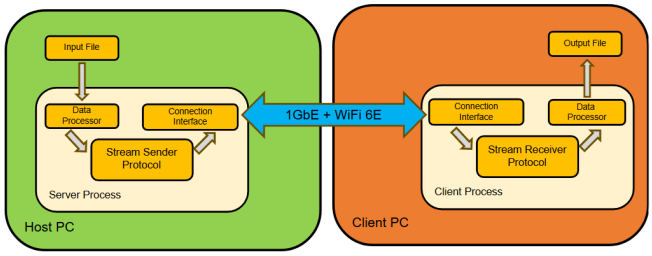

Fig. 10. Real world Testing Block Diagram

4.4 Challenges and Limitations

Throughout the evaluation, several challenges were encountered:

- **Debugging the Ethernet Subsystem:** In some cases, the FPGA was successfully transmitting packets, but the host PC failed to capture them, likely due to mismatches in signals such as TVALID and TREADY within the AXI stream between IP cores. This issue highlights the need for further debugging and optimization of the FPGA design.
- **Tool Limitations:** Standard benchmarking tools like iperf3 were not suitable for loopback testing with only one system due to reliance on the kernel network stack, which would not let the system send packets to itself through a NIC.

4.5 Summary of Evaluation Results

An important aspect of our evaluation was to study the impact of protocol parameters, notably sliding window size, on both throughput and reliability. Preliminary results indicate that while a larger window size can reduce the number of round-trip delays and increase throughput, it also introduces challenges in managing retransmissions efficiently. Our tests comparing raw TCP, UDP and our custom "Pseudo-TCP" approach demonstrate that pure TCP incurs slightly more overhead due to compared to our NACK approach, while UDP alone lacks reliability. The NACK-based sliding window protocol strikes a balance by providing error recovery without the latency penalties associated with TCP congestion control and slow start mechanisms.

Overall, the evaluation confirms that our custom streaming protocol is capable of delivering high-throughput data transmission with low latency, even under challenging conditions. Loopback tests achieved a measured throughput of 2.69 Gbps, and real-world LAN tests, despite exhibiting additional network-induced variability, demonstrated the protocol's robustness. These results validate the design choices made—particularly the use of a sliding window protocol combined with a NACK-based error recovery mechanism—and highlight areas for future improvement. Moving forward, the next steps include further debugging of the FPGA Ethernet subsystem, optimizing the memory offloading process, and porting the protocol to a full FPGA implementation for bidirectional streaming.

5 Conclusion

In this paper, we presented a novel UDP-based streaming protocol tailored for real-time RF data transmission between RFSoC platforms and host computers. By integrating a lightweight "pseudo-TCP" approach that employs Negative Acknowledgements (NACKs) and a sliding window mechanism, our design effectively balances the tradeoff between reliability and low latency - a critical requirement in modern SDR applications such as Radio Access Networks. Experimental evaluations using loopback testing have demonstrated our protocol shows promise in improving reliability while maintaining the low latency required for real-time streaming. These results confirm that our approach can support the stringent performance demands of emerging 5/6G systems and other high-bandwidth applications. Looking ahead, further work is needed in supporting ZeroMQ-based messaging on the host end, supporting bidirectionality, and implmenting streaming of real I/Q data, be it framed in VITA49.2 or a custom packet payload. Overall, our contributions underscore the potential of custom streaming protocols for bridging the gap between performant hardware and practical system demands.

6 Acknowledgement

Thank you to the Wireless Communications, Sensing, and Networking Group (WCSNG) at UC San Diego for allowing us to use their RFSoC and networking equipment, in addition to providing valuable insights. Special thanks to Agrim, Adel and Jeeva for their constant support and guidance.

7 Team Contribution

(1) Kishore - FPGA Setup, Worked on the Vivado block design from scratch and developed the custom Vitis HLS blocks, Graphics for documentation, SFP Testbed setup
(2) Russ - Interface Control Document and Initial C++ Codebase, Performed LAN and SFP+ tests, Documentation
(3) Daniel - Debugged and re-implemented Streaming Protocol, benchmarking scripts and code documentation, Helped with the HLS workflow and code development

All team members contributed significant amounts to critical components of the project.

8 Bibliography

References

[1] Daniel Ziper, Kishore Rajendran, R. S. rfsoc-streaming-257b. https://github.com/ece257b/rfsoc-streaming-257b, 2025. Accessed: 2025-01-24.
[2] García Redondo, M., et al. Rfsoc gen3-based sdr for bolometer readout. *Journal of Low Temperature Physics* (2024). Accessed: 2025-03-19.
[3] Gartmann, R., et al. Evaluating rfsoc as sdr for microcalorimeters. *JINST 19* (2024), C02078. Accessed: 2025-03-19.
[4] Guide, A. P. 10g/25g ethernet subsystem product guide.
[5] Gujarathi, V. Understanding streaming protocols. https://dev.to/, 2023. Accessed: 2025-03-19.
[6] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J., Roskind, J., Kulik, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., Chang, W.-T., and Shi, Z. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, Association for Computing Machinery, p. 183–196.
[7] Media, S. Media over quic – high-quality, low-latency streaming. https://www.streamingmedia.com/, 2023. Accessed: 2025-03-19.
[8] Noronha, C., et al. A study of protocols for low-latency video transport over the internet. In *BEIT Conference Proceedings* (2018). Accessed: 2025-03-19.
[9] Research, E. *UHD Manual: Device Streaming & Radio Transport Protocols*. Ettus Research, 2019. Accessed: 2025-03-19.
[10] Sahakyan, K. Streaming sensor/camera data using rdma: A paradigm shift beyond udp and tcp. Online, 2023. Accessed: 2025-03-19.

Received 20 March 2025